MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

E.T.S. INGENIEROS INFORMÁTICOS

OPTIMIZACIÓN DE ALGORITMOS PARA PROCESADO DE IMÁGENES CON GPUs

TRABAJO FIN DE MÁSTER

AUTOR

José Jaime González Itúrbide

TUTOR

Estíbaliz Martínez Izquierdo

RESUMEN

El objetivo de este proyecto es evaluar la mejora de rendimiento que aporta la paralelización de algoritmos de procesamiento de imágenes, para su ejecución en una tarjeta gráfica. Para ello, una vez seleccionados los algoritmos a estudio, fueron desarrollados en lenguaje C++ bajo el paradigma secuencial.

A continuación, tomando como base estas implementaciones, se paralelizaron siguiendo las directivas de la tecnología CUDA (Compute Unified Device Architecture) desarrollada por NVIDIA.

Posteriormente, se desarrolló un interfaz gráfico de usuario en Visual C#, para una utilización más sencilla de la herramienta.

Por último, se midió el rendimiento de cada uno de los algoritmos, en términos de tiempo de ejecución paralela y speedup, mediante el procesamiento de una serie de imágenes de distintos tamaños.

SUMMARY

The aim of this Project is to evaluate the performance improvement provided by the parallelization of image processing algorithms, which will be executed on a graphics processing unit. In order to do this, once the algorithms to study were selected, each of them was developed in C++ under sequential paradigm.

Then, based on these implementations, these algorithms were implemented using the compute unified device architecture (CUDA) programming model provided by NVIDIA.

After that, a graphical user interface (GUI) was developed to increase application's usability.

Finally, performance of each algorithm was measured in terms of parallel execution time and speedup by processing a set of images of different sizes.

Tabla de contenido

LISTADO DE FIGURAS

LISTADO DE TABLAS

1. INTRODUCCIÓN

Hoy en día el procesamiento de imágenes obtenidas mediante teledetección se aplica en numerosos ámbitos (agricultura, reconocimiento de zonas, detección de terrenos y aguas contaminadas, evaluación de desastres naturales,…).

Debido a la criticidad de algunos de los ámbitos mencionados y la gran resolución de las imágenes satelitales, uno de los grandes retos en el entorno de la teledetección es la implementación de algoritmos rápidos y eficientes que reduzcan considerablemente el tiempo de procesamiento y se asemejen a un sistema de tiempo real.

Es por ello que el objetivo de este trabajo se centrará en el análisis de rendimiento de CUDA (Compute Unified Device Architecture), afamada arquitectura presente en las tarjetas gráficas NVIDIA muy utilizada actualmente.

Dicho análisis se realizará mediante la implementación de varios algoritmos en dos paradigmas distintos: secuencial y paralelo, para su posterior aplicación sobre un número concreto de imágenes de distintos tamaños y por último, la obtención de una serie de índices que permitirán su comparación.

2. ESTADO DEL ARTE

2.1. FUNDAMENTOS TEÓRICOS

2.1.1. TELEDETECCIÓN

Se puede definir formalmente teledetección o detección remota, del inglés 'remote sensing', como "*la ciencia y/o el arte de adquirir información sin contacto directo entre el captador y el objetivo*".

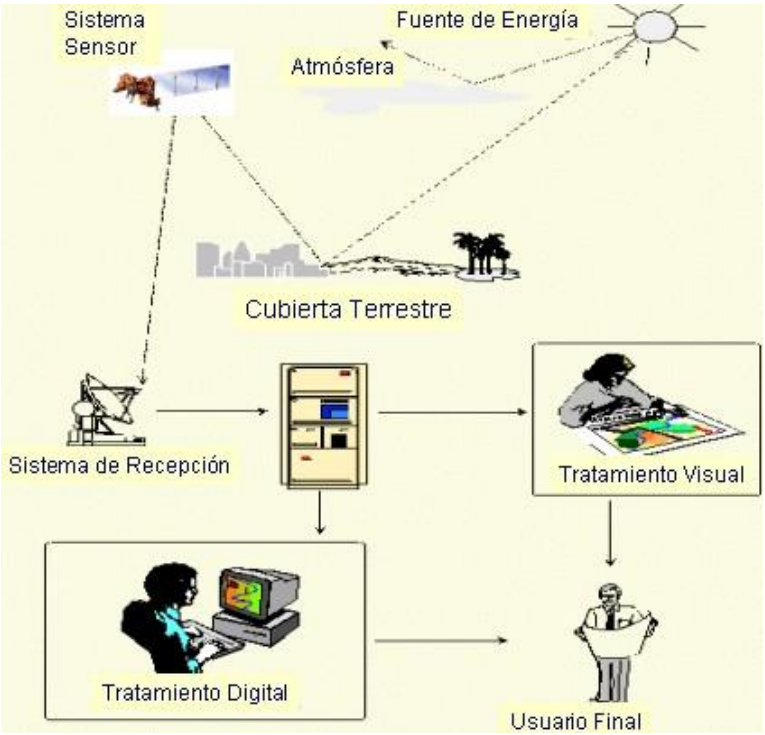

Figura 1. Definición de teledetección. Fuente: uncoma

Como se puede observar en la figura 1, la teledetección se basa en la obtención de información emitida o reflejada por la superficie de la Tierra, a través de instrumentos de grabación, de escaneo o simplemente que no están en contacto directo con el objeto, para su posterior tratamiento y análisis.

La teledetección es una herramienta o técnica basada en aspectos comunes a las Matemáticas o la Física. Los sensores detectan la radiación electromagnética que sale de un objeto o área geográfica desde una distancia y extraen la información evaluable usando algoritmos basados en matemáticas y estadística, por lo que se puede considerar como una actividad científica.

Para que la observación sea posible es necesario que, aunque sin contacto material, exista algún tipo de interacción entre los objetos y el sensor. En este caso, la interacción va a ser un flujo de radiación que parte de los objetos y se dirige hacia el sensor. Este flujo puede ser, en cuanto a su origen, de tres tipos:

- Radiación solar reflejada por los objetos (luz visible e infrarrojo reflejado).
- Radiación terrestre emitida por los objetos (infrarrojo térmico).
- Radiación emitida por el sensor y reflejada por los objetos (radar).

Las técnicas basadas en los dos primeros tipos se conocen como teledetección pasiva y la última de ellas como teledetección activa:

- **Teledetección pasiva:** En la teledetección pasiva se detecta la radiación natural emitida o reflejada por el objeto o área de estudio.
- **Teledetección activa:** Por el contrario, en la teledetección activa es el propio teledetector el que emite su propio haz de radiación para posteriormente recoger su respuesta.

2.1.2. IMAGEN DIGITAL

Una imagen digital es una representación bidimensional de una imagen a partir de una matriz numérica, frecuentemente en codificación discretizada. Dependiendo de la resolución de la imagen pueden clasificarse en:

- **Imagen matricial**: Una imagen matricial, también conocida como imagen ráster, imagen en mapa de bits o imagen bitmap, es una estructura o fichero de datos que representa una rejilla rectangular de píxeles o puntos de color, denominada matriz, que se puede visualizar en un monitor, papel u otro dispositivo de representación. Estas imágenes se definen por su altura y anchura, ambas en píxeles, y por su profundidad de color, en bits por píxel, que determina el número de colores distintos que se pueden almacenar en cada punto individual, y por lo tanto, en gran medida, la calidad de la imagen. Este formato está ampliamente extendido y es el que se emplea normalmente en la toma de fotografías digitales, así como para realizar capturas de vídeo. En dicha toma o captura se utilizan dispositivos de conversión analógica-digital, tales como escáneres y cámaras digitales.

  Cada punto representado en la imagen debe contener información de color, representada en canales separados que simbolizan los componentes primarios del tono que se pretende reproducir en cualquier modelo de color, bien sea RGB (rojo-verde-azul, del inglés red-green-blue), CMYK (cyan-magenta-yellow-key, en castellano cian-magenta-amarillo-negro, llamado key en este modelo), LAB (nombre abreviado de dos espacios de color diferentes, CIELAB y Hunter Lab, ya que ambos están relacionados) o cualquier otro modelo disponible para su representación. A esta información, se le puede sumar otro canal que escenifica la transparencia respecto al fondo de la imagen. En algunos casos, como es el de GIF (Graphics Interchange Format o Formato de Intercambio de Gráficos) el canal de transparencia tiene un

solo bit de información, es decir, se puede representar como totalmente opaco o como totalmente transparente; en los más avanzados, como es el caso de PNG (Portable Network Graphics o Gráficos de Red Portátiles) y TIFF (Tagged Image File Format o Formato de Fichero de Imagen Etiquetada), el canal de transparencia tiene la misma profundidad que los canales de color, con lo que se pueden obtener centenares, miles o incluso millones de niveles de transparencia distintos.

Al cambiar las dimensiones de una imagen matricial se puede vislumbrar una notoria pérdida de calidad. Esta desventaja contrasta con las posibilidades que ofrecen los gráficos vectoriales, que pueden adaptar su resolución fácilmente a la de cualquier dispositivo de visualización. Existe una pérdida mayor cuando se pretende incrementar el tamaño de la imagen (aumentar la cantidad de píxeles por lado) que cuando se efectúa una reducción del mismo.

- **Gráfico vectorial**: Un gráfico o imagen vectorial es una imagen digital formada por objetos geométricos independientes (segmentos, polígonos, arcos...), cada uno de ellos definido por distintos atributos matemáticos de forma, de posición, de color, etc. Por ejemplo, un círculo quedaría definido por la posición de su centro, su radio, el grosor de la línea y su color.

  El interés principal de los gráficos vectoriales es poder ampliar el tamaño de una imagen a voluntad sin sufrir la pérdida de calidad que sufren los mapas de bits. De la misma forma, permiten mover, estirar y retorcer imágenes de manera relativamente sencilla. Su uso también está muy extendido en la generación de imágenes en tres dimensiones tanto estáticas como dinámicas.

  Las principales aplicaciones de los gráficos vectoriales son:

  - *Generación de gráficos:* Se utilizan para crear logos ampliables tanto como se quiera, así como en el diseño técnico con programas de tipo CAD (Computer Aided Design, o Diseño Asistido por Ordenador). Muy populares para generar escenas 3D.
  - *Lenguajes de descripción de documentos:* Los gráficos vectoriales permiten describir el aspecto de un documento independientemente de la resolución del dispositivo de salida. Los formatos más conocidos son PostScript y PDF (Portable Document Format, o Formato de Documento Portátil). A diferencia de las imágenes matriciales, se puede visualizar e imprimir estos documentos en cualquier resolución sin ningún tipo de pérdida.
  - *Tipografías:* Casi todas las aplicaciones actuales usan texto formado por imágenes vectoriales. Los ejemplos más comunes son TrueType, OpenType y PostScript.

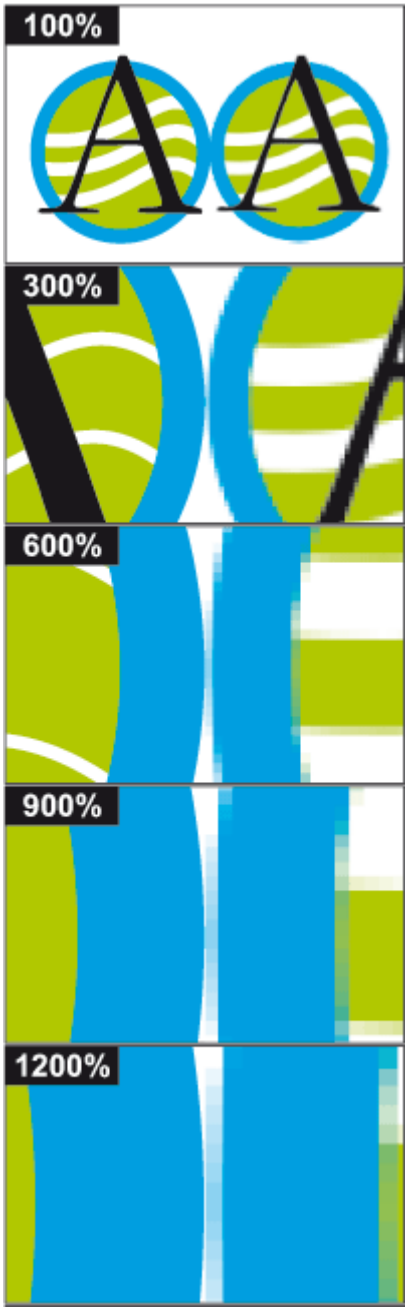

Figura 2. Gráfico vectorial vs Imagen matricial. Fuente: Wikipedia

- *Videojuegos:* Los gráficos vectoriales se utilizan con mucha frecuencia para videojuegos con gráficos en tres dimensiones.
- *Internet:* Los gráficos vectoriales que se encuentran en la red suelen ser o bien de formatos abiertos como VML (Vector Markup Language, Lenguaje de Marcado de Vectores) y SGV (Scalable Vector Graphics o Gráficos Vectoriales Escalables) o SWF (Small Web Format o Formato Web Pequeño) en formato propietario. Estos últimos se pueden visualizar con Adobe Flash Player.

En la figura 2, se pueden comparar los gráficos vectoriales (columna izquierda) con las imágenes matriciales (columna de la derecha) al ampliar las respectivas imágenes. Como se puede observar, a medida que aumenta el zoom los gráficos de la izquierda mantienen su calidad, mientras que los situados a la derecha van revelando paulatinamente los píxeles que forman la imagen. Los gráficos vectoriales pueden ser escalados ilimitadamente sin que se aprecie pérdida de calidad. Los dos ejemplos de ampliación al 300% y 600% ilustran en gran manera esta propiedad de los mencionados gráficos vectoriales: los contornos de las figuras geométricas (franjas blancas detrás de la letra A) no aumentan proporcionalmente en la figura en el caso del gráfico ráster o imagen matricial.

2.1.2.1. IMÁGENES TIFF

TIFF (Tagged Image File Format) es un formato de archivo para guardar imágenes vectoriales creado por Aldus en 1986, empresa que posteriormente fue adquirida por Adobe Systems, la cual es propietaria del formato actualmente.

Se trata de un formato de imagen con etiquetas, las cuales permiten almacenar información de las características de la imagen, lo que favorece a posteriori el procesado de la misma.

Estas etiquetas comprenden el número de filas, el número de columnas, el número de bits que se ha utilizado para codificar un píxel, el tipo de compresión, el número de muestras por píxel (etiqueta muy importante para el uso de imágenes multiespectrales), etc.

Para el manejo de este tipo de imágenes se ha utilizado, en este trabajo, la librería "libtiff", que provee una serie de herramientas que permiten tanto la lectura como la escritura de dichas imágenes de varias formas posibles.

2.2. FUNDAMENTOS TECNOLÓGICOS

2.2.1. MICROSOFT VISUAL STUDIO 2010

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para sistemas operativos Windows. Trabaja con numerosos lenguajes de programación, entre los que destacan C++, C#, Visual Basic .Net, F#, Java, Python, Ruby y PHP. Asimismo, soporta entornos de desarrollo web como ASP.NET, MVC, Django,…

En Visual Studio se pueden crear aplicaciones que se comuniquen entre estaciones de trabajo, páginas web, dispositivos móviles, sistemas empotrados, consolas, etc.

2.2.2. C/C++

C++ es un lenguaje de programación de propósito general. Se trata de un lenguaje multiparadigma, ya que puede utilizarse bajo el paradigma genérico, así como el estructurado y el orientado a objetos e incluye herramientas de manipulación de objetos a bajo nivel.

Inicialmente fue desarrollado por Bjarne Stroustrup en los laboratorios Bell, comenzando en 1979. Su objetivo era obtener un lenguaje flexible y eficiente (como C), pero que además suministrara herramientas de alto nivel para facilitar la organización de los programas.

Se diseñó para favorecer la programación de sistemas (por ejemplo, sistemas empotrados o núcleos de sistemas operativos) en términos de rendimiento, eficiencia y flexibilidad de uso. Existen implementaciones de C++ disponibles en varias plataformas y es suministrado por diferentes organizaciones, entre las que destacan Microsoft e Intel.

C++ está estandarizado por la International Organization of Standardization (ISO) desde 1998; la última versión ratificada y publicada por la ISO, a Junio de 2015, es la de Diciembre de 2014 (ISO/IEC 14882:2014, conocida informalmente como C++14).

Lenguajes de programación como C#, Java y las versiones más recientes de C se han visto influenciados por C++.

2.2.3. VISUAL C#

Microsoft Visual C# es un lenguaje de programación diseñado para crear una gran variedad de aplicaciones que se ejecutan en .NET Framework. C# es simple, eficaz, con seguridad de tipos y orientado a objetos. Con diversas innovaciones, C# permite desarrollar aplicaciones rápidamente y mantiene la expresividad y elegancia de los lenguajes de tipo C.

Visual Studio admite Visual C# con un editor de código completo, plantillas de proyecto, diseñadores, asistentes para código, un depurador eficaz y fácil de usar, además de otras herramientas. La biblioteca de clases .NET Framework ofrece acceso a una amplia gama de servicios de sistema operativo y a otras clases útiles y adecuadamente diseñadas que aceleran el ciclo de desarrollo de manera significativa.

2.2.4. DRAG AND DROP

Drag and drop (arrastrar y soltar) es una funcionalidad utilizada en los interfaces de usuario gráficos que permite el desplazamiento de información a/entre aplicaciones arrastrando el elemento con el cursor del ratón y soltándolo encima de su destino.

Una operación Drag and Drop se desarrolla en cuatro etapas:

- Mover el puntero al objeto.
- Pulsar y mantener el botón del ratón u otro dispositivo puntero para agarrar el objeto.
- Arrastrar el objeto hasta la localización deseada moviendo el puntero hacia la misma.
- Soltar el objeto dejando de pulsar el botón.

Al soltar el elemento hay dos posibilidades:

- Se desplaza el elemento arrastrado por el ratón. En este caso, desaparece del control de partida al soltarlo encima del control de destino.
- Se copia el elemento arrastrado. En este proceso, se suelta una copia sobre el control de destino.

2.2.5. CUDA

CUDA son las siglas de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo). Se trata de una arquitectura de cálculo paralelo, creada por NVIDIA, que pone a disposición de los desarrolladores un compilador y un conjunto de herramientas que les permite codificar algoritmos en una Unidad de Procesamiento Gráfico (GPU, de sus siglas en inglés) usando una variación del lenguaje de programación C.

Introducida en noviembre de 2006, se basa en la utilización de un elevado número nodos de procesamiento para realizar operaciones en paralelo sobre un gran volumen de datos bajo la arquitectura SIMT (Sigle Instruction Multiple Thread), similar al paradigma SIMD (Single Instruction Multiple Data), obteniendo una considerable reducción del tiempo de procesamiento y un gran aumento de las prestaciones.

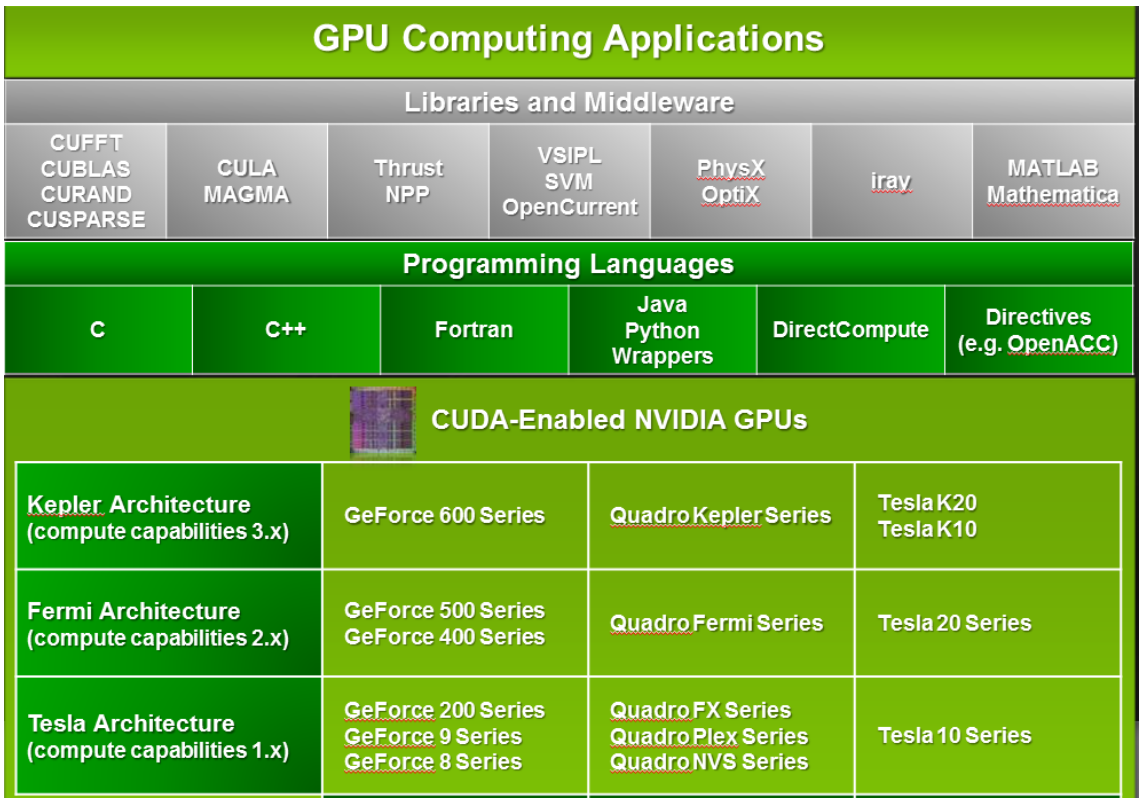

Figura 3. Aplicaciones de cómputo en GPU. Fuente: NVIDIA

Para desarrollar aplicaciones en CUDA, se pueden utilizar lenguajes de programación como C, C++ y Fortran, entre otros. También existen un conjunto de librerías y middlewares como se muestra en la figura 3, en la que también se incluyen las distintas arquitecturas CUDA con sus capacidades de cómputo y los conjuntos de GPU que se encuentran en las mismas.

2.2.5.1. VENTAJAS DE CUDA

Las ventajas de CUDA con respecto a otros tipos de computación sobre GPU que utilizan API gráficas son:

- Lecturas dispersas: Se puede consultar cualquier posición de memoria.
- Memoria virtual unificada (desde CUDA 4.0).
- Memoria unificada (desde CUDA 6.0)
- Lecturas más rápidas desde y hacia la GPU.
- Soporte para enteros y operadores a nivel de bit.
- Memoria compartida: CUDA posee una zona de memoria que puede compartirse entre hilos. Se trata de un área de unos pocos kilobytes (KB) que puede ser utilizada como cache debido a su rapidez.

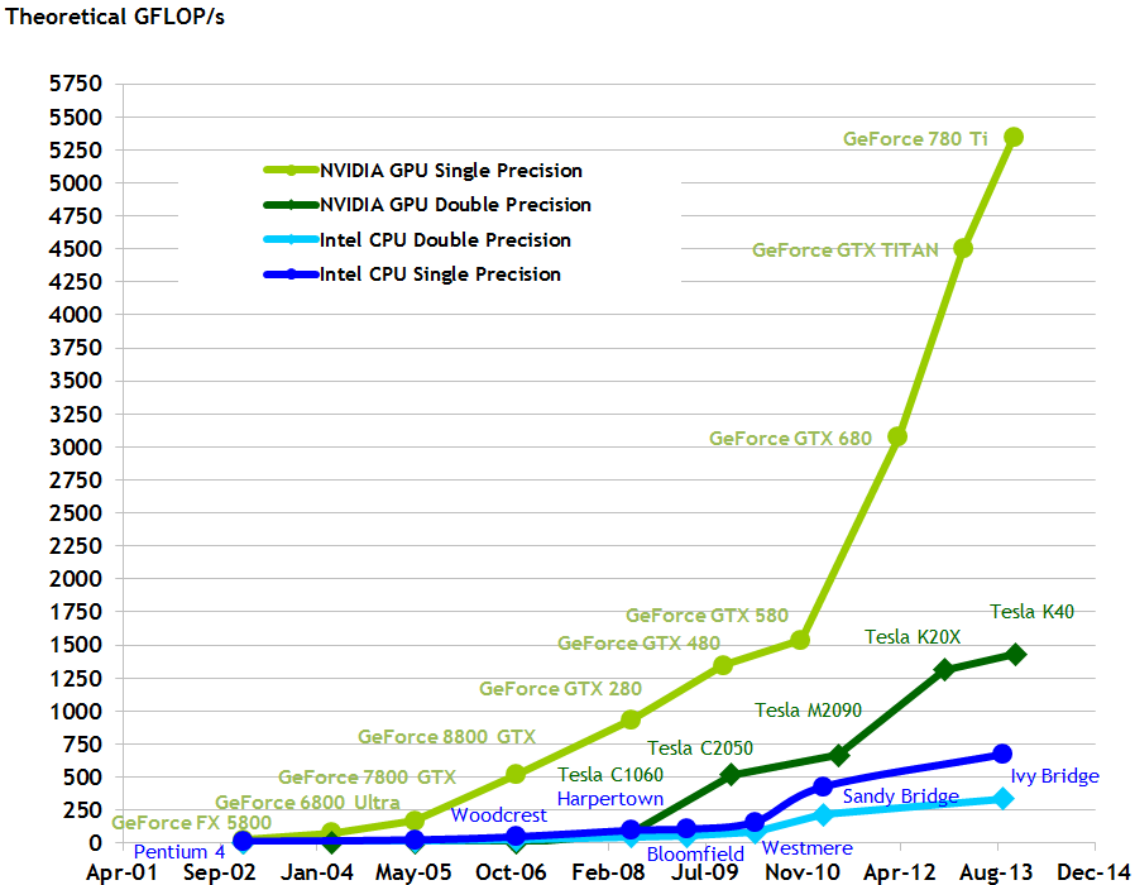

Figura 4. Operaciones de coma flotante por segundo para CPU y GPU. Fuente: NVIDIA

Los beneficios que aporta el uso de esta tecnología se intuyen en las figuras 4 y 5, donde se comparan el número de operaciones en coma flotante y el ancho de banda de memoria entre la CPU y la GPU respectivamente.

En ambos gráficos se puede observar que tanto el número de operaciones en coma flotante como el ancho de banda de memoria han sufrido un crecimiento exponencial con el lanzamiento de las nuevas GPU de arquitectura Kepler. No obstante, se puede intuir que, a lo largo de los años, las prestaciones de la GPU son siempre muy superiores a las de una CPU convencional.

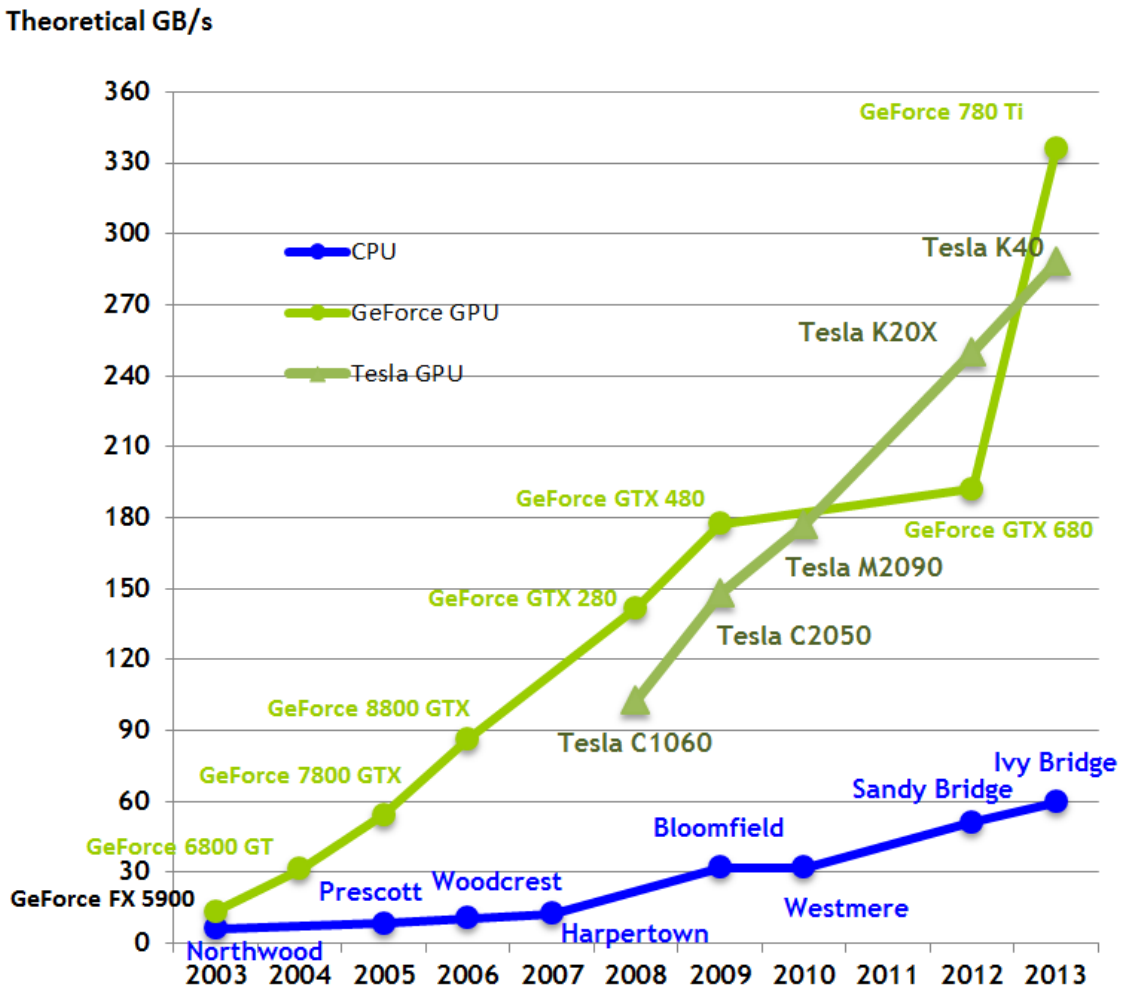

Figura 5. Ancho de banda de memoria para CPU y GPU. Fuente: NVIDIA

2.2.5.2. LIMITACIONES DE CUDA

CUDA también presenta una serie de limitaciones que se exponen a continuación:

- No es compatible con todo el estándar C: Imposibilidad de utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable.
- Renderizado de texturas no soportado.
- No soporta números no normalizados (NaN) en precisión simple en la primera generación CUDA (dispositivos con capacidad de cómputo 1.x).
- El ancho de banda de los buses y sus latencias pueden provocar un cuello de botella entre la Unidad de Central de Procesamiento (CPU) y la GPU.
- Manejo de excepciones no soportado por temas de rendimiento.
- Por razones de eficiencia. Los hilos deben lanzarse en grupos de al menos 32.

2.2.5.3. ELEMENTOS NECESARIOS PARA DESARROLLAR EN CUDA

Para desarrollar aplicaciones con arquitectura CUDA se necesita una tarjeta gráfica NVIDIA compatible. Actualmente existe una gran cantidad de modelos y productos compatibles con esta tecnología, los cuales se pueden dividir en los siguientes grupos:

- CUDA–Enabled Tesla Products
- CUDA–Enabled Quadro Products
- CUDA–Enabled NVS Products
- CUDA–Enabled GeForce Products
- CUDA–Enabled TEGRA/Jetson Products

CUDA puede trabajar tanto en sistemas operativos Windows como en Unix. Sin embargo, existe una serie de características no soportadas, entre las que se encuentran:

- Sistemas Linux x86 32 bits.
- Red Hat Enterprise Linux 5 y CentOs 5.
- A partir de la versión 7.0, CUDA Toolkit y CUDA Driver no son compatibles con las arquitecturas sm_10, sm_11, sm_12 y sm_13.
- Los sistemas operativos Windows de 32 bits no son compatibles con CUDA Toolkit. Asimismo, en los sistemas operativos Windows de 64 bits, CUDA Toolkit y CUDA Driver no son compatibles con las siguientes características:
    - Ejecución de aplicaciones de 32 bits en los productos Tesla y Quadro.
    - Uso de la librería Thrust desde aplicaciones de 32 bits.
    - Versiones de 32 bits de las librerías científicas de CUDA Toolkit, incluyendo cuBLAS, cuSPARSE, cuFFT, cuRAND y NPP.
    - Versiones de 32 bits de los ejemplos de CUDA.
- Uso de gcc como compilador del sistema operativo en Mac OS X.

Una vez que se dispone de un sistema operativo y una tarjeta gráfica compatibles con CUDA, el siguiente paso es instalar el paquete de herramientas proporcionado por NVIDIA.

Lo primero que se debe realizar es la actualización de los drivers de la tarjeta gráfica. Para ello se debe acceder a la página web de NVIDIA y en la sección de descargas seleccionar el modelo de la tarjeta gráfica que se va a utilizar, así como el sistema operativo. Una vez descargados, instalarlos y comprobar que no se producen errores en dicha instalación.

Seguidamente, se procede a la descarga de CUDA Toolkit (conjunto de herramientas que permite desarrollar en CUDA). En la página web de NVIDIA se encuentran todas las versiones del mismo. Lo ideal es descargarse la última versión disponible, pero es importante cerciorarse de la compatibilidad tanto de la tarjeta gráfica como del sistema que se van a utilizar con dicha versión.

Cabe destacar que los paquetes de instalación se dividen según el sistema operativo que se utilice. Asimismo, en versiones previas a CUDA 7, los paquetes de instalación para ordenadores portátiles son distintos a los de sobremesa, por lo que es muy importante elegir el instalador adecuado.

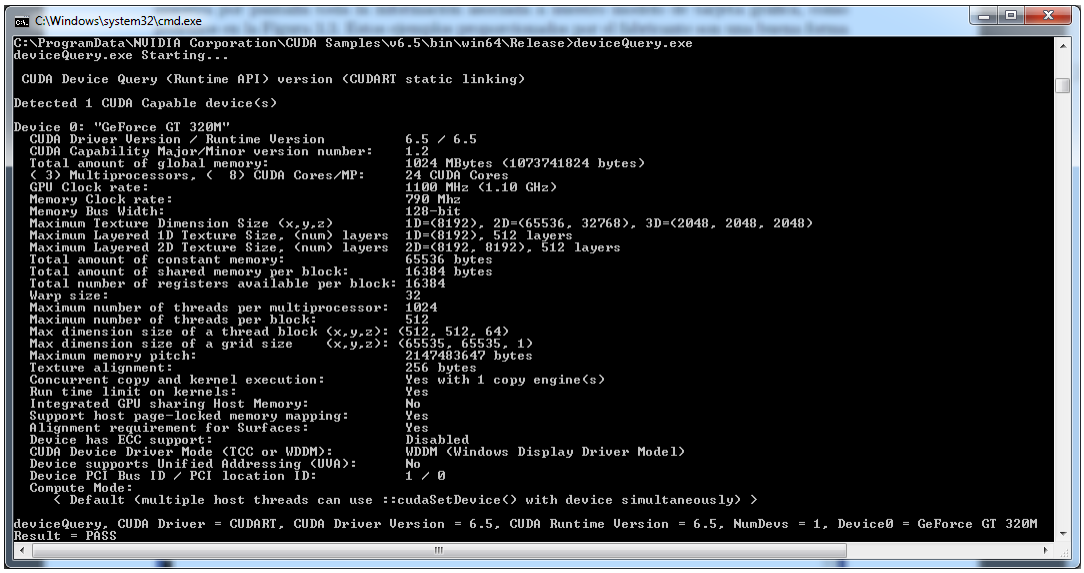

Figura 6. Ejecución de un ejemplo del SDK de CUDA. Fuente: Propia

El Software Development Kit (SDK) de CUDA contiene una serie de ejemplos que es aconsejable utilizar para verificar la correcta instalación del CUDA Toolkit así como el apropiado funcionamiento de la tarjeta gráfica. Entre ellos se encuentra "deviceQuery", que muestra información de la GPU albergada en el sistema (figura 6).

Una vez esté todo instalado correctamente, para implementar una aplicación en CUDA, se necesitan al menos un fichero .c o .cpp, dependiendo de si se desarrolla en C o C++, que contendrá el código que se ejecutará en la CPU, y otro .cu, donde se alojará la parte del código que se ejecutará en paralelo en la GPU.

Cada una de las extensiones mencionadas anteriormente se compila con un compilador distinto. El compilador nvcc, proporcionado por NVIDIA se encarga de la compilación de los ficheros .cu, para realizar la separación del código que se va a ejecutar en la CPU del que lo va a hacer en el GPU.

Por su parte, el compilador gcc, proporcionado en el estándar C, se encarga de la compilación de la parte que se va a ejecutar en la CPU, así como de establecer los enlaces con los archivos objeto (incluyendo el generado por nvcc) y generar el archivo ejecutable.

2.2.5.4. CONCEPTOS BÁSICOS DE CUDA

En esta sección, se van a exponer algunos de los conceptos más característicos de CUDA, que servirán para conocer un poco mejor el lenguaje que se utiliza en el desarrollo de aplicaciones paralelas.

Hay tres conceptos que es importante tener muy claros a la hora de implementar una aplicación paralela: thread, thread block y grid.

- Thread (Hilo): Es cada uno de los hilos de control que van a ejecutar simultáneamente una función o kernel.
- Thread block (Bloque de hilos): Conjunto de threads lanzados sobre el mismo multiprocesador de la GPU. Cada uno de los threads contenidos en un bloque puede tener comunicación con el resto a través del área de memoria compartida.
- Grid: Representa todo el conjunto de bloques de threads. Los bloques de threads pueden comunicarse entre sí mediante la memoria global.

Otro concepto muy importante es el de kernel. Un kernel es una subrutina paralela que se ejecuta sobre CUDA para todos los threads de un grid. Se define mediante la declaración __global__ y el número de threads que ejecuta el kernel para una llamada al mismo se especifica mediante la nueva sintaxis de configuración de la ejecución <<<…>>>.

Cada thread que ejecuta el kernel tiene un único identificador de hilo, accesible desde el kernel a través de la variable threadIdx.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figura 7. Ejemplo de función kernel. Fuente: NVIDIA

En la figura 7 se muestra un ejemplo de una función kernel, que realiza la suma de dos vectores de tamaño N (A y B) y deja el resultado en C. En dicho ejemplo, cada uno de los N threads que ejecuta VecAdd() realiza una suma.

Por convención, threadIdx es un vector de 3 componentes, para que los hilos puedan ser identificados usando un índice de una, dos o tres dimensiones, formando un bloque de threads de una, dos y tres

dimensiones respectivamente. De esta manera, se proporciona una manera natural para invocar elementos tales como vector, matriz o matriz tridimensional.

El número de threads por bloque está limitado, ya que se espera que todos los threads de un bloque residan en el mismo núcleo del procesador y deben compartir los limitados recursos de memoria del core. En la GPU actuales, un thread block puede contener hasta 1024 hilos.

Sin embargo, se puede ejecutar un kernel mediante múltiples bloques de threads, por lo que el número total de threads es el número de threads por bloque multiplicado por el número de bloques.

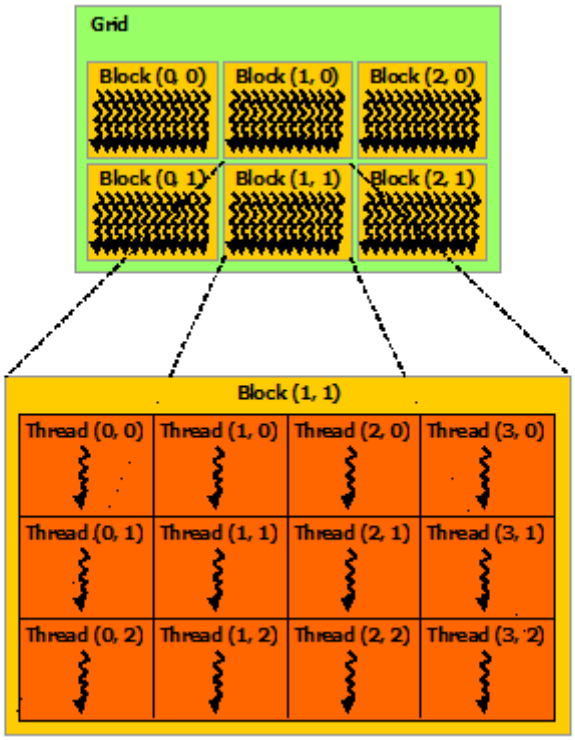

Figura 8. Grid de bloques de threads. Fuente: NVIDIA

Los bloques se organizan en un grid de una, dos o tres dimensiones, como se puede observar en la figura 8. El número de threads en un grid viene dado normalmente por el tamaño de los datos a procesar o el número de procesadores del sistema.

Cada bloque del grid se identifica por un índice de una, dos o tres dimensiones accesible desde el kernel a través de la variable blockIdx. Las dimensiones del bloque de threads se encuentran en la variable blockDim.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figura 9. Ejemplo de código para sumar dos matrices. Fuente: NVIDIA

La figura 9 contiene un código en el que se advierte como se realiza el acceso a los elementos de las matrices mediante la suma del identificador del thread con la multiplicación del identificador del bloque y el tamaño del mismo.

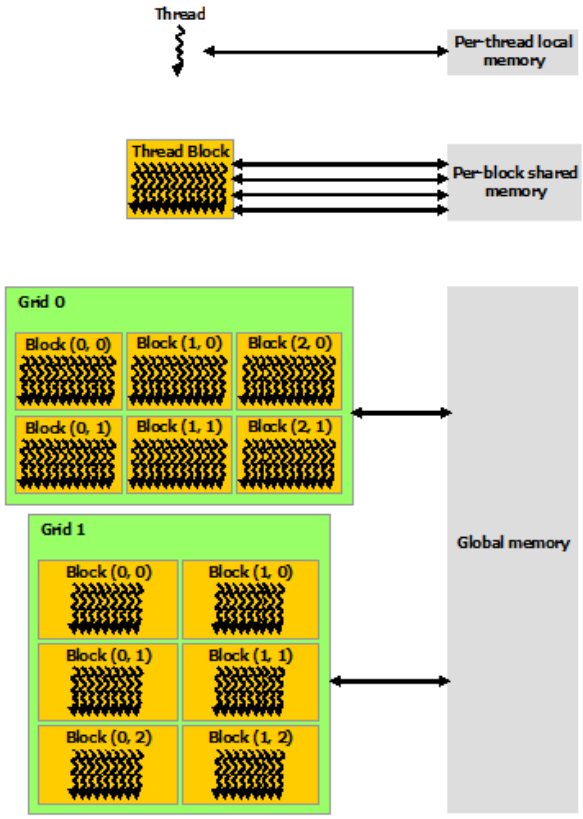

Figura 10. Jerarquía de memoria en CUDA. Fuente: NVIDIA

Los threads pueden acceder a los datos desde múltiples espacios de memoria durante la ejecución (figura 10). Cada uno de los threads tiene su espacio de memoria local. Adicionalmente, cada thread tiene memoria compartida visible para el resto de threads del bloque disponible durante el ciclo de vida del bloque. Por último, todos los hilos de un grid tienen acceso a la misma memoria global.

A su vez, existen dos zonas de memoria de sólo lectura adicionales accesibles desde cualquier thread: constante y textura. Estas dos zonas junto con la memoria global son persistentes a lo largo de las llamadas al kernel desde la misma aplicación.

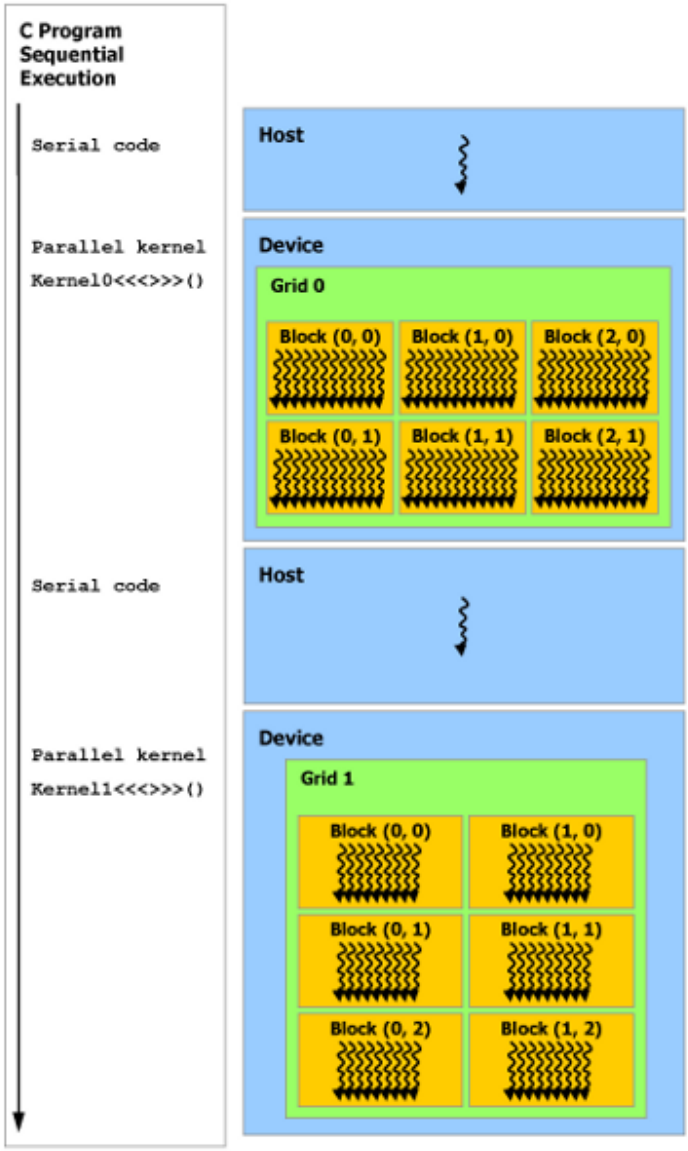

Figura 11. Proceso de ejecución de una aplicación CUDA. Fuente: NVIDIA

Como se muestra en la figura 11, el modelo de programación de CUDA asume que los hilos ejecutan en un dispositivo físicamente separado que actúa como coprocesador del host que ejecuta el programa desarrollado en C. Este caso se da cuando el kernel ejecuta el código en la GPU y el resto del programa C lo hace en la CPU.

El modelo de programación de CUDA también asume que el host y el dispositivo tienen espacios de memoria diferentes.

Asimismo, en la figura 11 se puede observar que cuando el programa se ejecuta en el host lo hace de manera secuencial, mientras que cuando se trata de la GPU lo hace en paralelo.

2.2.5.5. PLATAFORMA DE DESARROLLO UTILIZADA

En la realización de este trabajo se ha utilizado el modelo GeForce GT 320M de NVIDIA.

CARACTERÍSTICA	VALOR
Capacidad de computación	1.2
Memoria global	1024 MB
Número de multiprocesadores	24
Frecuencia de reloj de la GPU	1.1 GHz
Memoria constante	65536 bytes
Memoria compartida por bloque	16384 bytes
Número máximo de hilos por bloque	512
Número máximo de bloques por grid	65535

Tabla 1. Características de la GPU NVIDIA GeForce GT 320M.

Se trata de una GPU de la primera generación de CUDA, cuyas características se incluyen en la tabla 1. Es fundamental tener en cuenta los valores número máximo de hilos por bloque y número máximo de bloques por grid, ya que, en caso de superarlos, se pueden obtener errores durante la ejecución de la aplicación o resultados erróneos.

Otro dato a tener en cuenta es que la memoria global es de 1 GB, por lo que habrá que pasar los datos por partes a la GPU en la mayoría de las ocasiones. Esto es debido a que las imágenes que se usan en teledetección suelen ser de un tamaño mayor que 1 GB.

3. DESARROLLO

3.1. DESCRIPCIÓN DE LA HERRAMIENTA

La herramienta implementada consta de una serie de algoritmos de tratamiento de imágenes TIFF. Con una interfaz amigable basada en Visual C#, utiliza CUDA para la ejecución paralela de las instrucciones sobre los píxeles de la imagen, así como la librería "libtiff" para la lectura, desde la imagen de entrada y escritura, en la imagen de salida (también en formato TIFF), de los mismos.

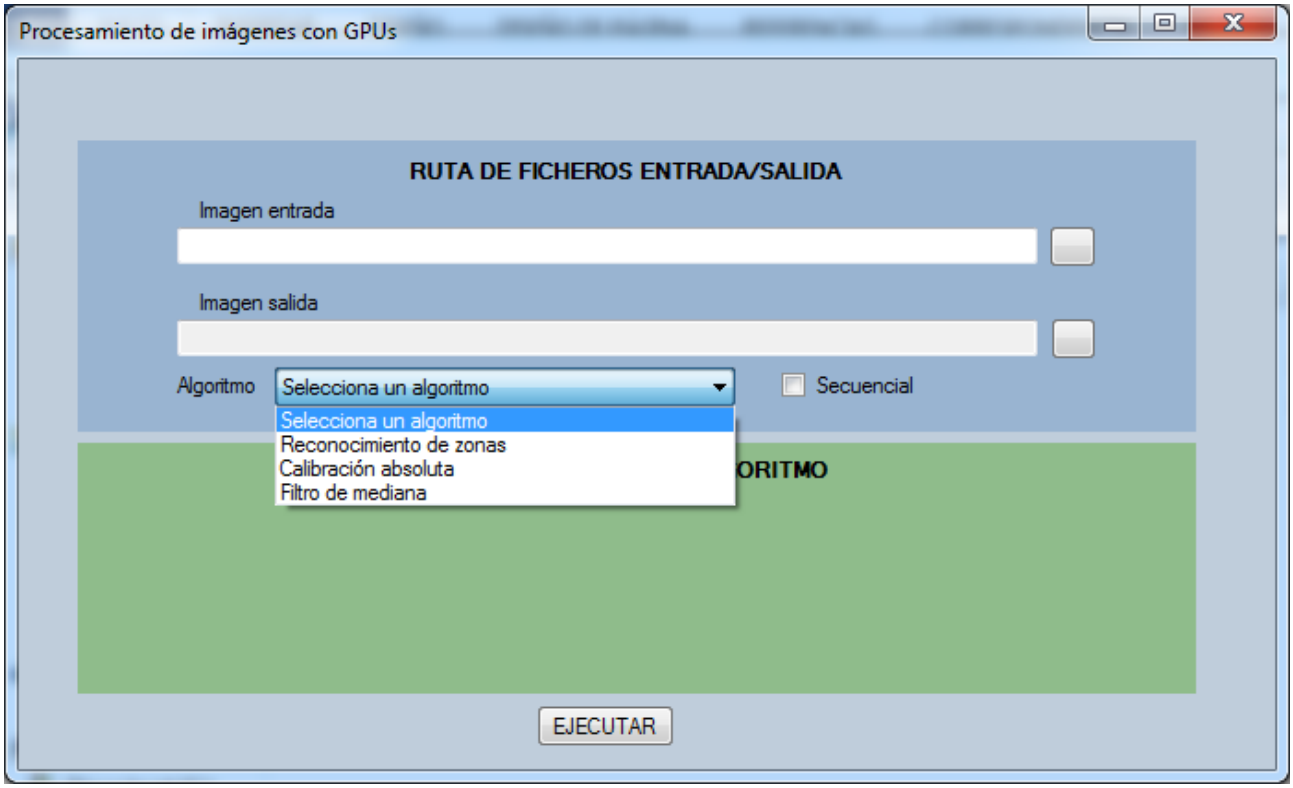

Figura 12. Pantalla principal de la herramienta. Fuente: Propia

Como se puede comprobar en la figura 12, perteneciente a la pantalla principal del interfaz de la herramienta, se han desarrollado los siguientes algoritmos, los cuales se resumirán más adelante:

- Algoritmo de reconocimiento de zonas.
- Algoritmo de calibración absoluta.
- Algoritmo filtro de mediana.

Se han implementado dos versiones de cada algoritmo; en primer lugar una versión secuencial, es decir, sin paralelismo (elegible mediante el check box "Secuencial"), y una vez verificado el correcto funcionamiento de la misma, se ha implementado la versión paralela tomando como base la anterior.

Este es el fundamento de los desarrollos paralelos y el objetivo que persigue este trabajo, la implementación de dos versiones para poder comparar los tiempos de procesamiento de las mismas entre sí. Obviamente, esta tarea se realiza únicamente en la fase de pruebas con un conjunto de datos de tamaño reducido, ya que de tratarse de un conjunto de datos excesivamente grande, la ejecución de la aplicación secuencial puede llevar días, meses e incluso años, como por ejemplo, el renderizado de películas de animación.

3.2. INSTRUCCIONES DE USO

Figura 13. Ejemplo de ejecución de la herramienta. Fuente: Propia

Los pasos a seguir durante la ejecución de la herramienta son:

- Selección del algoritmo que se quiere ejecutar.
- Introducción de los parámetros requeridos.
- Una vez seleccionados todos los parámetros, pulsar el botón “EJECUTAR”, para comenzar el procesamiento de la imagen.
- Esperar a que finalice la ejecución del programa.

En la figura 13 se muestra un ejemplo de ejecución del algoritmo de calibración absoluta.

3.3. ALGORITMOS IMPLEMENTADOS

En esta sección se va a realizar un estudio más detallado de los algoritmos mencionados anteriormente.

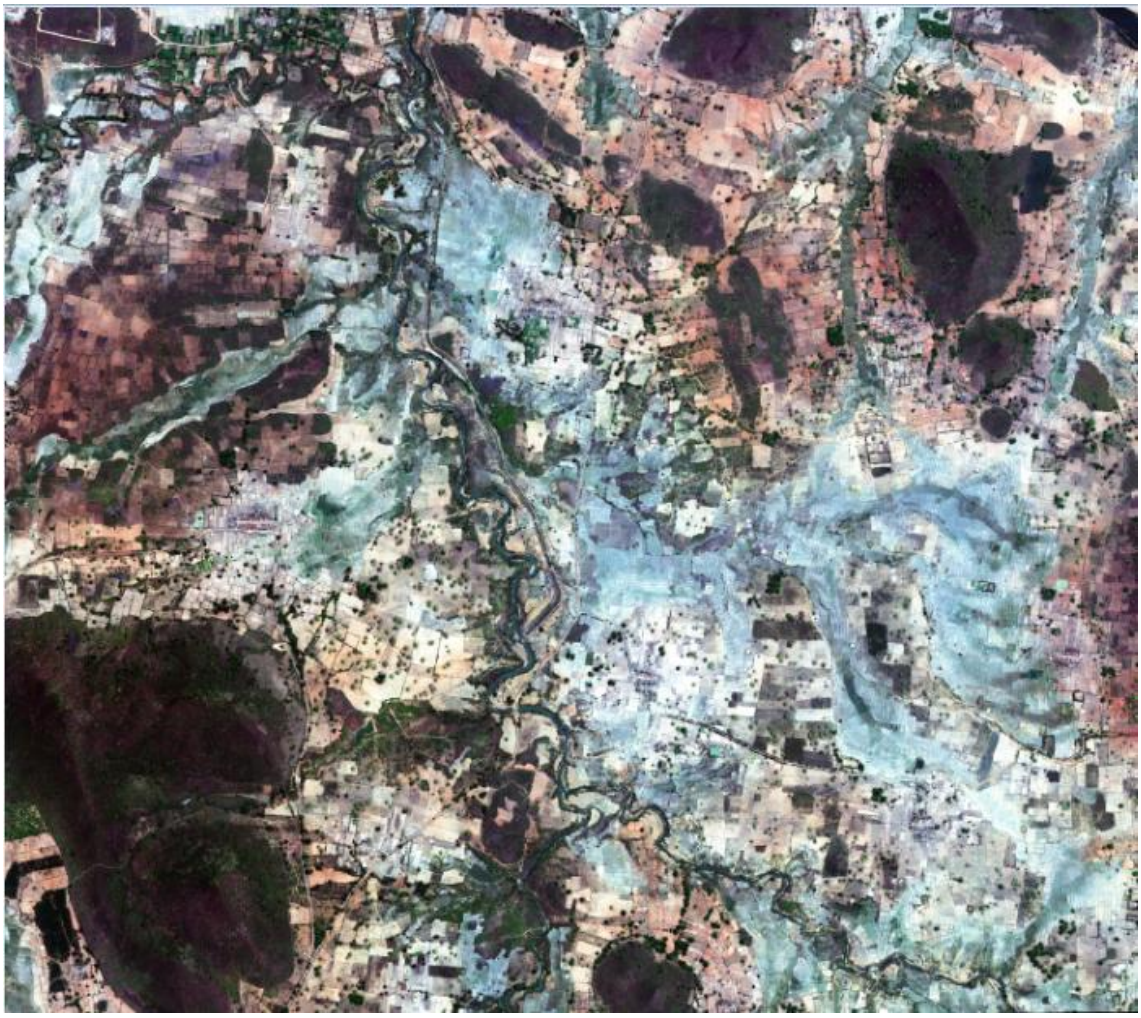

Figura 14. Ejemplo de imagen de entrada

Para una mejor ilustración del cometido de los algoritmos, se proporcionará un ejemplo de imagen de salida para cada uno de los mismos basado en la imagen de entrada expuesta en la figura 14.

3.3.1. ALGORITMO DE RECONOCIMIENTO DE ZONAS (ARZ)

3.3.1.1. PARÁMETROS DE ENTRADA

Figura 15. Interfaz de selección de parámetros para el algoritmo de reconocimiento de zonas. Fuente: Propia

El algoritmo de reconocimiento de zonas recibe los parámetros expuestos en la figura 15, que pertenece al interfaz gráfico de la herramienta. Dichos parámetros representan:

- **Imagen entrada**: Ruta de la imagen TIFF en color.
- **Imagen salida**: Ruta donde se quiere almacenar el fichero con la imagen resultante.
- **Banda R**: Se introducirán:
  - **Umbral superior**: Límite superior del valor del píxel de la banda roja. Valor entre 0 y 255.
  - **Umbral inferior**: Límite inferior del valor del píxel de la banda roja. Valor entre 0 y 255.
- **Banda G:** Se introducirán:
  - **Umbral superior**: Límite superior del valor del píxel de la banda verde. Valor entre 0 y 255.
  - **Umbral inferior**: Límite inferior del valor del píxel de la banda verde. Valor entre 0 y 255.
- **Banda B:** Se introducirán:
  - **Umbral superior**: Límite superior del valor del píxel de la banda azul. Valor entre 0 y 255.
  - **Umbral inferior**: Límite inferior del valor del píxel de la banda azul. Valor entre 0 y 255.
- **Umbral color superior**: Color en formato RGB que se asignará a los píxeles cuyo valor se encuentre por encima de umbral superior.
- **Umbral color inferior**: Color en formato RGB que se asignará a los píxeles cuyo valor se encuentre por debajo de umbral inferior.

3.3.1.2. PARÁMETROS DE SALIDA

Seguidamente se incluye la lista de parámetros de salida de este algoritmo:

- **Fichero con la imagen de salida**: Se guarda en la ruta especificada por el usuario.
- **Tiempo de ejecución total.**
- **Tiempo de ejecución del fragmento de código paralelo.**

3.3.1.3. LÓGICA DE FUNCIONAMIENTO

El algoritmo de reconocimiento de zonas obtendrá, en la imagen de salida, las áreas de píxeles cuyos valores para cada una de las bandas, se encuentren dentro de sus respectivos rangos.

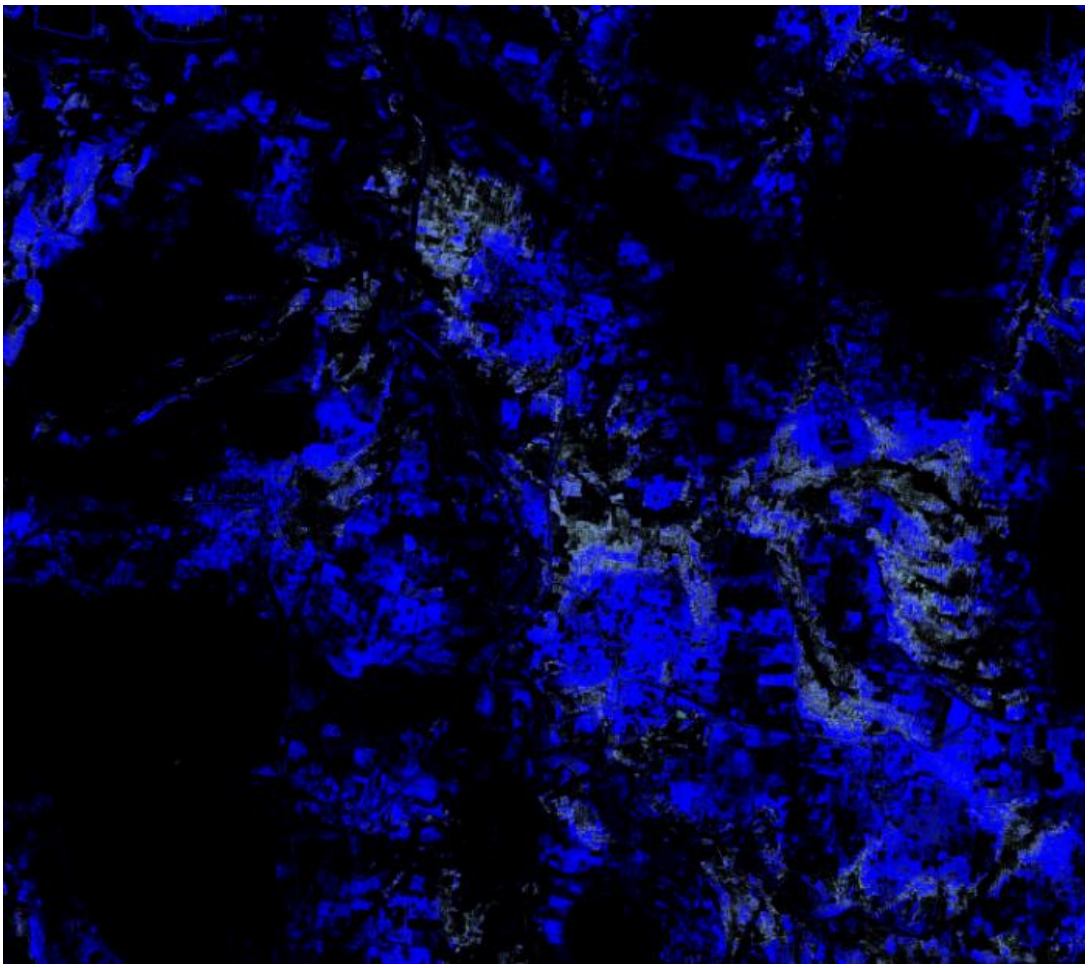

Figura 16. Ejemplo de imagen de salida del algoritmo de reconocimiento de zonas. Fuente: Propia

Dicho de otro modo, por cada píxel, se comprobará su pertenencia al rango especificado para cada una de las bandas (roja, verde y azul) y sí y sólo sí el píxel cumple la pertenencia a todas ellas, se mostrará en su color original en la imagen de salida, si, por el contrario, su valor es mayor que el umbral superior o menor que el umbral inferior, se le asignará el color seleccionado por el usuario para sendos casos. Todo esto se puede verificar en la figura 16, en la que se ha proporcionado aleatoriamente al programa los umbrales para la imagen de entrada y los colores para la imagen de salida mencionados a continuación:

- **Umbral inferior rojo**: 105
- **Umbral superior rojo**: 185
- **Umbral inferior verde**: 198
- **Umbral superior verde**: 238
- **Umbral inferior azul**: 192
- **Umbral superior azul**: 255
- **Color para valor superior al rango**: Azul (0, 0, 255)
- **Color para valor inferior al rango**: Negro (0, 0, 0)

3.3.1.4. CÓDIGO

```
uint32 i;
for (i = 0; i < size; i++) {
		if (v1[i] < uinf1 || v2[i] < uinf2 || v3[i] < uinf3) {
			v1[i] = color_inf[0];
			v2[i] = color_inf[1];
			v3[i] = color_inf[2];
		}
		else if (v1[i] > usup1 || v2[i] > usup2 || v3[i] > usup3) {
			v1[i] = color_sup[0];
			v2[i] = color_sup[1];
			v3[i] = color_sup[2];
		}
}
```

Figura 17. Fragmento de código secuencial del algoritmo de reconocimiento de zonas. Fuente: Propia

```
uint32 i = threadIdx.x + blockIdx.x * blockDim.x;
if (i < size[0]) {
	if (v1[i] < uinf1[0] || v2[i] < uinf2[0] || v3[i] < uinf3[0]) {
		v1[i] = color_inf[0];;
		v2[i] = color_inf[1];;
		v3[i] = color_inf[2];;
	}
	else if (v1[i] > usup1[0] || v2[i] > usup2[0] || v3[i] > usup3[0]) {
		v1[i] = color_sup[0];;
		v2[i] = color_sup[1];;
		v3[i] = color_sup[2];;
	}
}
```

Figura 18. Fragmento de código paralelo del algoritmo de reconocimiento de zonas. Fuente: Propia

En este apartado se muestran las diferencias entre dos fragmentos de código, el primero perteneciente a la implementación secuencial del algoritmo (figura 17), mientras que el de la figura 18 pertenece a la paralela.

En la implementación secuencial se observa el uso de un bucle "for" que recorre todos los píxeles de la imagen.

Por su parte, la paralela posee como característica principal la utilización del identificador del thread, obtenido a partir de la suma del índice del mismo (threadIdx.x) con el producto del índice del bloque (blockIdx.x) y el tamaño del propio (blockDim.x). Nótese que al tratarse de matrices de una sola dimensión (vectores), se ha usado únicamente la componente x de cada uno de los elementos.

Otro detalle a destacar es que todos los parámetros que recibe la función implementada en el algoritmo paralelo son vectores, ya que es la estructura usada por CUDA para copiarlos de la memoria de la CPU a su homónima en la GPU.

3.3.2. ALGORITMO DE CALIBRACIÓN ABSOLUTA (ACA)

3.3.2.1. PARÁMETROS DE ENTRADA

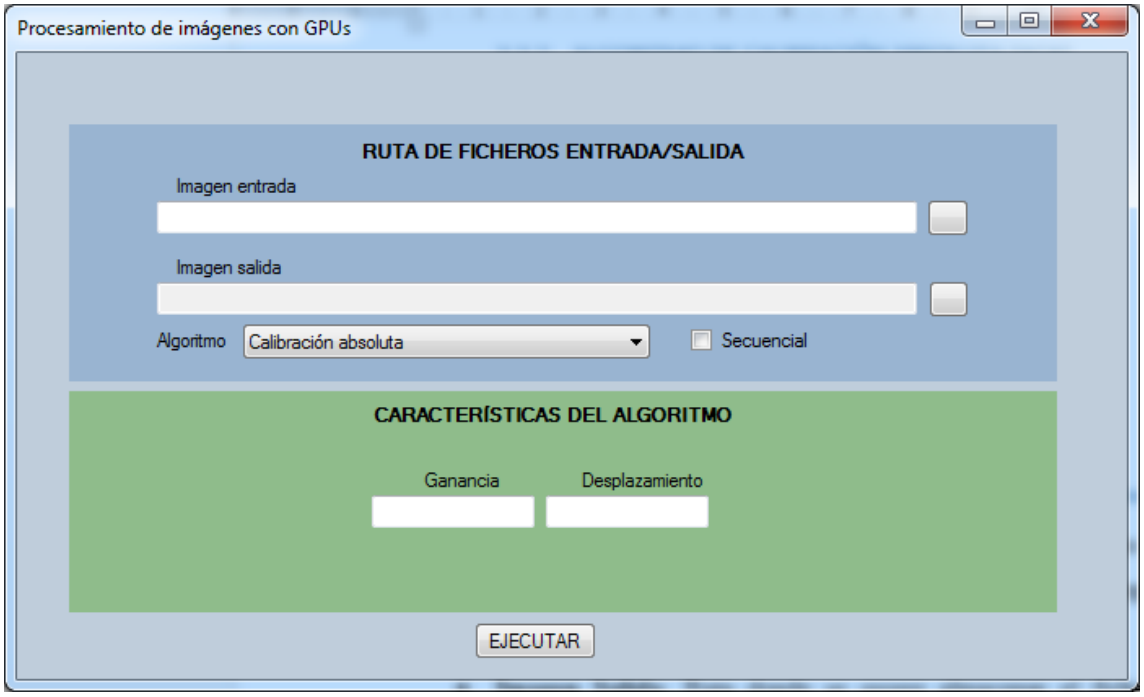

Figura 19. Interfaz de selección de parámetros para el algoritmo de calibración absoluta. Fuente: Propia

El algoritmo de calibración absoluta recibe los parámetros mostrados en la figura 19, que pertenece al interfaz gráfico de la herramienta. Dichos parámetros representan:

- **Imagen entrada**: Ruta de la imagen TIFF en color.
- **Imagen salida**: Ruta donde se quiere almacenar el fichero con la imagen resultante.

- **Ganancia**: Ganancia que se quiere aplicar a los píxeles de la imagen de salida. Valor entre 1 y 10.
- **Desplazamiento**: Desplazamiento que se aplicará a cada píxel una vez aplicada la ganancia. Valor entre 0 y 10.

3.3.2.2. PARÁMETROS DE SALIDA

Seguidamente se incluye la lista de parámetros de salida de este algoritmo:

- **Fichero con la imagen de salida**: Se guarda en la ruta especificada por el usuario.
- **Tiempo de ejecución total.**
- **Tiempo de ejecución del fragmento de código paralelo.**

3.3.2.3. LÓGICA DE FUNCIONAMIENTO

La misión del algoritmo de calibración absoluta es convertir los píxeles de la imagen de unidades digitales a unidades físicas. Este proceso se consigue mediante el suministro de una ganancia a la imagen, para el caso de este trabajo se ha establecido la siguiente fórmula:

$$píxel_{salida} = píxel_{entrada} \times ganancia + desplazamiento$$

Además, como medida de prevención, en caso de que el píxel de salida supere el valor máximo establecido por la resolución radiométrica de la imagen, se le asignará dicho valor máximo.

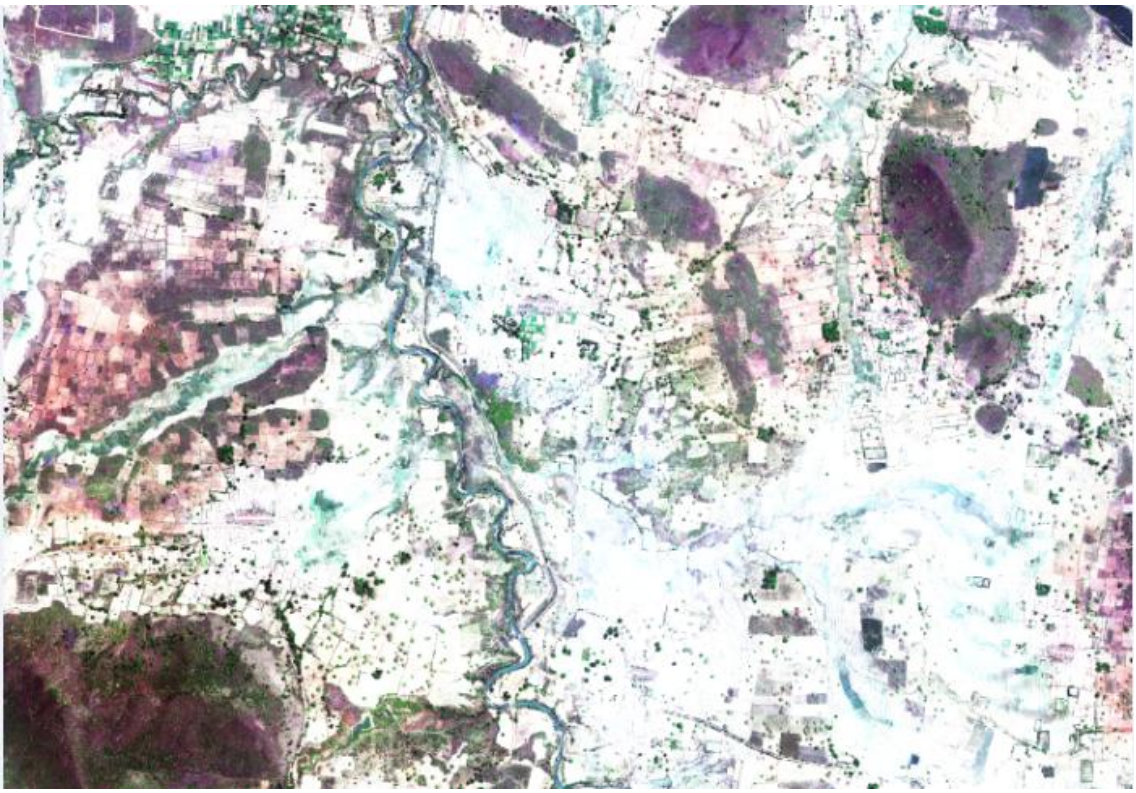

Figura 20. Ejemplo de imagen de salida del algoritmo de calibración absoluta. Fuente: Propia

La figura 20 muestra un ejemplo de imagen calibrada obtenida mediante la aplicación totalmente aleatoria de una ganancia de 2 y un desplazamiento de 6 a la imagen original.

3.3.2.4. CÓDIGO

```
uint8 saturacion (int pixel) {
	if (pixel > MAX_VALUE) {
		return (uint8) MAX_VALUE;
	}
	return (uint8) pixel;
}

void calibracion8 (uint8* v, uint32 size, int gain, int offset) {
	uint32 i;
	for (i = 0; i < size; i++) {
		v[i] = saturacion8(v[i] * gain + offset);
	}
}
```

Figura 21. Fragmento de código secuencial del algoritmo de calibración absoluta. Fuente: Propia

```
__device__ uint8 saturacion8 (int pixel) {
	if (pixel > MAX_VALUE) {
		return (uint8) MAX_VALUE;
	}
	return (uint8) pixel;
}

__global__ void calibra8 (uint8* v, uint32* size, int* gain, int* offset) {
	uint32 i = threadIdx.x + blockIdx.x * blockDim.x;

	if (i < size[0]) {
		v[i] = saturacion8(v[i] * gain[0] + offset[0]);
	}
}
```

Figura 22. Fragmento de código paralelo del algoritmo de calibración absoluta. Fuente: Propia

Examinando las figuras 21 (algoritmo secuencial) y 22 (algoritmo paralelo), se vuelven a presentar las diferencias observadas en el algoritmo de reconocimiento de zonas:

- Uso del identificador de thread en lugar del bucle "for" para el acceso a los píxeles de la imagen.
- Todos los parámetros se obtienen a través de punteros para el caso del algoritmo paralelo.

Asimismo, se han adjuntado las definiciones de las funciones de ambos algoritmos en sendos fragmentos de código. Centrándose en el fragmento de código paralelo, se observa el empleo de unas palabras reservadas en CUDA delante del tipo de dato que devuelve la función, se trata de los calificadores de tipos de función.

El calificador __device__ indica que la función se ejecutará en la GPU y únicamente puede llamarse desde la misma.

A su vez, el calificador __global__ señala que la función se ejecuta en la GPU, puede llamarse desde la CPU y desde la GPU, este último únicamente para capacidades de cómputo superiores a 3.0. Además, el tipo que devuelve una función __global__ es obligatoriamente "void", es decir, no puede devolver ningún parámetro.

3.3.3. ALGORITMO FILTRO DE MEDIANA

3.3.3.1. PARÁMETROS DE ENTRADA

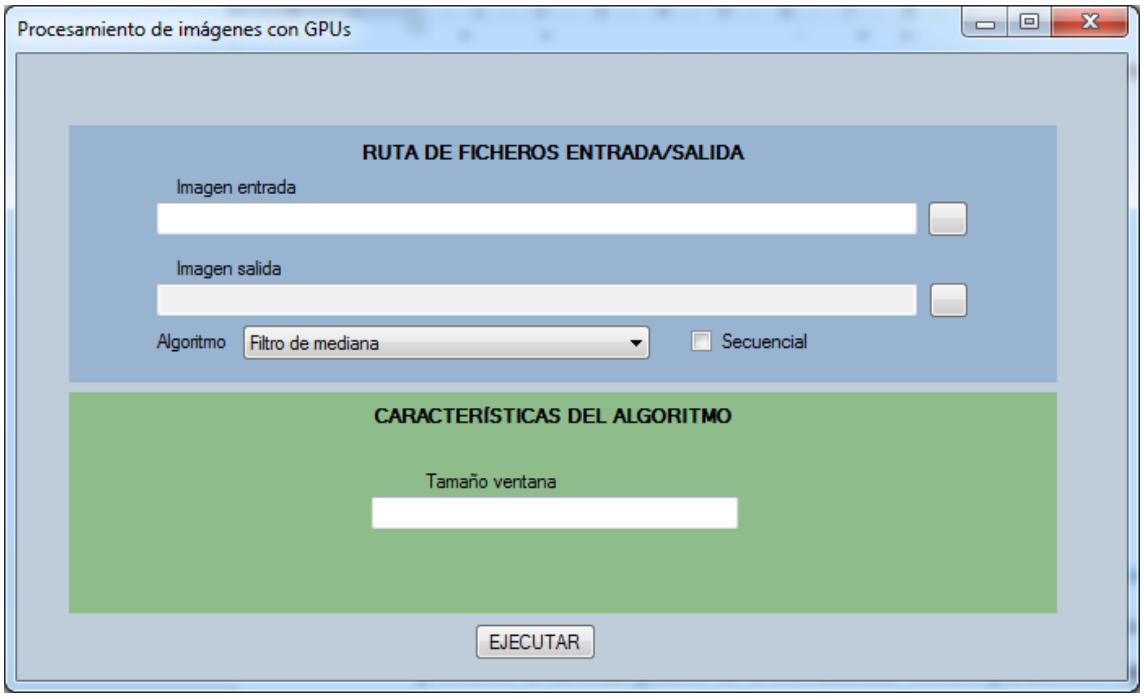

Figura 23. Interfaz de selección de parámetros para el algoritmo filtro de mediana. Fuente: Propia

El algoritmo de calibración absoluta recibe los parámetros mostrados en la figura 23, que pertenece al interfaz gráfico de la herramienta. Dichos parámetros representan:

- **Imagen entrada**: Ruta de la imagen TIFF en color.
- **Imagen salida**: Ruta donde se quiere almacenar el fichero con la imagen resultante.
- **Tamaño ventana**: Ancho de la ventana que utilizará la herramienta para calcular el valor de cada píxel. Valores permitidos: 3, 5 o 7.

3.3.3.2. PARÁMETROS DE SALIDA

Al igual que en los algoritmos anteriores, los parámetros de salida de éste son:

- **Fichero con la imagen de salida**: Se guarda en la ruta especificada por el usuario.
- **Tiempo de ejecución total.**
- **Tiempo de ejecución del fragmento de código paralelo.**

3.3.3.3. LÓGICA DE FUNCIONAMIENTO

El cometido de este algoritmo es aplicar un filtro de mediana a cada uno de los píxeles de la imagen. Para realizar esta tarea, por cada píxel se debe ordenar todos los píxeles contenidos en la ventana por su valor y escoger el que se encuentre en la posición central, o lo que es lo mismo, encontrar el valor que cumple que la mitad de los valores de la muestra son superiores a él, y la otra mitad son inferiores al mismo. En comparación con otro tipo de filtros, la aplicación del filtro de mediana a una imagen es un proceso lento.

56	70	63
65	**61**	73
74	77	74

Tabla 2. Ejemplo de ventana 3x3

56	61	63	65	**70**	73	74	74	77

Tabla 3. Ordenación y resolución de un ejemplo con tamaño de ventana de 3x3

Suponiendo un tamaño de ventana de 3x3 con los valores de los píxeles presentados en la tabla 2, donde se resalta en color rojo el píxel cuyo valor se quiere calcular, se debe escoger el que se encuentre en quinto lugar tras la ordenación, como se muestra, resaltado en color verde, en la tabla 3.

3.3.3.4. CÓDIGO

```
void inserta_orden (uint8 elem, uint8* v, int cuantos) {
        int i = 0;
        int j;
        while (i < cuantos && elem > v[i]) {
                i++;
        }
        for (j = cuantos; j > i; j--) {
                v[j] = v[j - 1];
        }
        v[i] = elem;
}
```

Figura 24. Fragmento de código secuencial del algoritmo filtro de mediana. Fuente: Propia

```
__device__ void inserta_orden (uint8 elem, uint8* v, int pos_ini, int cuantos) {
        int i = pos_ini;
        int j;
        while (i < pos_ini + cuantos && elem > v[i]) {
                i++;
        }
        for (j = pos_ini + cuantos; j > i; j--) {
                v[j] = v[j - 1];
        }
        v[i] = elem;
}
```

Figura 25. Fragmento de código paralelo del algoritmo filtro de mediana. Fuente: Propia

Las figuras 24 y 25 contienen el código secuencial y paralelo respectivamente de la función que ordena el valor de los píxeles contenidos en la ventana.

En este caso, sólo se observa una diferencia, la introducción de un argumento más para el caso del código paralelo, que hace referencia a la posición inicial dentro del vector de ordenación.

Esto es debido a la creación únicamente de un vector de ordenación en la memoria de la GPU, con el tamaño necesario para albergar las ordenaciones de tantos píxeles como se puedan procesar al mismo tiempo, por lo que, para que no existan solapamientos entre threads, cada uno de ellos tiene una posición de inicio determinada, es decir, si se pueden procesar n píxeles simultáneamente, dentro del vector de ordenación existen n subvectores de tamaño ventana * ventana.

Por su parte, en el secuencial, al realizarse el procesamiento píxel a píxel, solamente se necesita un vector de ordenación de tamaño ventana * ventana.

3.4. CAMPOS DE APLICACIÓN

Existen diversos campos en los que se pueden aplicar los algoritmos implementados.

El algoritmo de reconocimiento de zonas sirve para encontrar superficies que cumplan la pertenencia en todas las bandas a los respectivos umbrales pasados como parámetro, por lo que tiene una gran variedad de aplicaciones, como por ejemplo, el descubrimiento de zonas habitadas en áreas recónditas para poder estudiar la instalación de luz, elementos necesarios para hacer llegar el agua potable, etc.

A su vez, el algoritmo de calibración absoluta sirve, como se mencionó anteriormente, para transformar las unidades digitales en unidades físicas, a partir de las cuales se pueden obtener otras magnitudes físicas como las radiancias.

Por su parte, el filtro de mediana se puede aplicar, como otra gran cantidad de filtros, para realizar el suavizado de la imagen y la eliminación de posibles ruidos de la misma. Aunque es un algoritmo de procesamiento lento, se obtienen buenos resultados tras su aplicación.

4. RESULTADOS

Esta sección engloba los resultados obtenidos para cada uno de los algoritmos.

Figura 26. Primera imagen sobre la que se aplican los algoritmos. Fuente: bitpalast.net

Figura 27. Imagen dos sobre la que se han aplicado los algoritmos. Fuente: zeiss.es

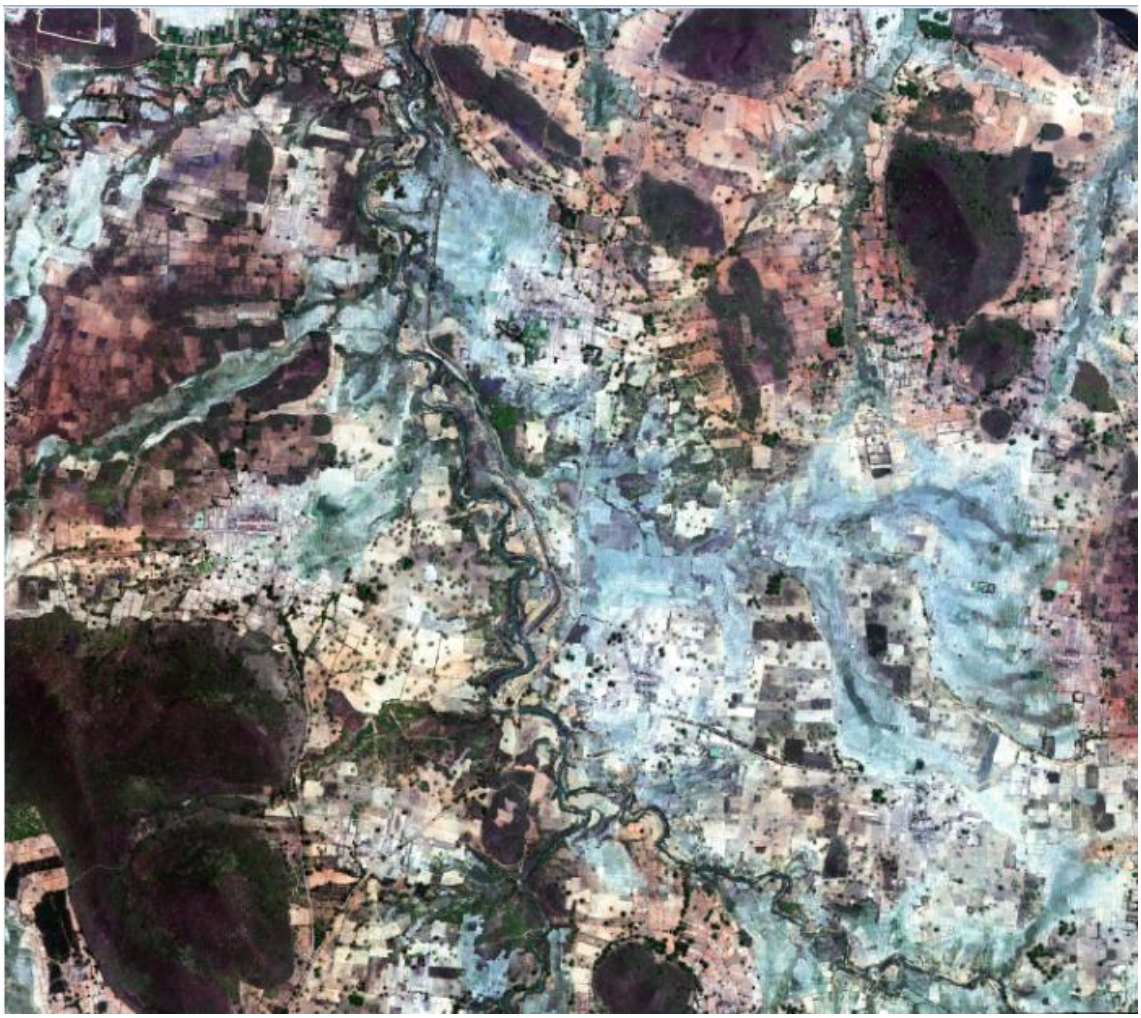

Figura 28. Imagen 3 sobre la que se han aplicado los algoritmos

Para la obtención de los mismos, se han procesado los algoritmos sobre cuatro imágenes de diferentes tamaños, para, de esta manera, poder medir el rendimiento de dichos algoritmos en cada una de ellas.

Imagen	Resolución	Total píxeles	Muestras/píxel	Tamaño (MB)
Imagen 1	1400x719	1.006.600	3	1,69
Imagen 2	4256x2832	12.052.992	4	47,7
Imagen 3	11762x10472	123.171.664	3	352
Imagen 4	23524x10472	246.343.328	3	704

Tabla 4. Características de las imágenes procesadas

Las tres primeras de ellas son las correspondientes a las figuras 26, 27 y 28 respectivamente, mientras que la última es el doble de la tercera, ya que se trata de la concatenación de dicha imagen en su ancho. Alguna de las características de estas imágenes que se muestran en la tabla 4. Se ha decidido usar imágenes de varios tamaños para, de esta forma, poder comparar el rendimiento de ambos paradigmas (secuencial y paralelo) en los distintos algoritmos.

4.1. ALGORITMO DE RECONOCIMIENTO DE ZONAS (ARZ)

RESOLUCIÓN DE LA IMAGEN	TIEMPO DE PROCESAMIENTO (S)		SPEEDUP
	CPU	GPU	CPU/GPU
1400x719	0,016	0,109	**0,147x**
4256x2832	0,202	0,187	1,080x
11762x10472	2,449	1,216	2,014x
23524x10472	4,602	2,215	**2,078x**

Tabla 5. Tiempos obtenidos para el algoritmo de reconocimiento de zonas

Los tiempos obtenidos para la ejecución del algoritmo de reconocimiento de zonas, tanto en su versión paralela (GPU) como en la secuencial (CPU) se muestran en la tabla 5.

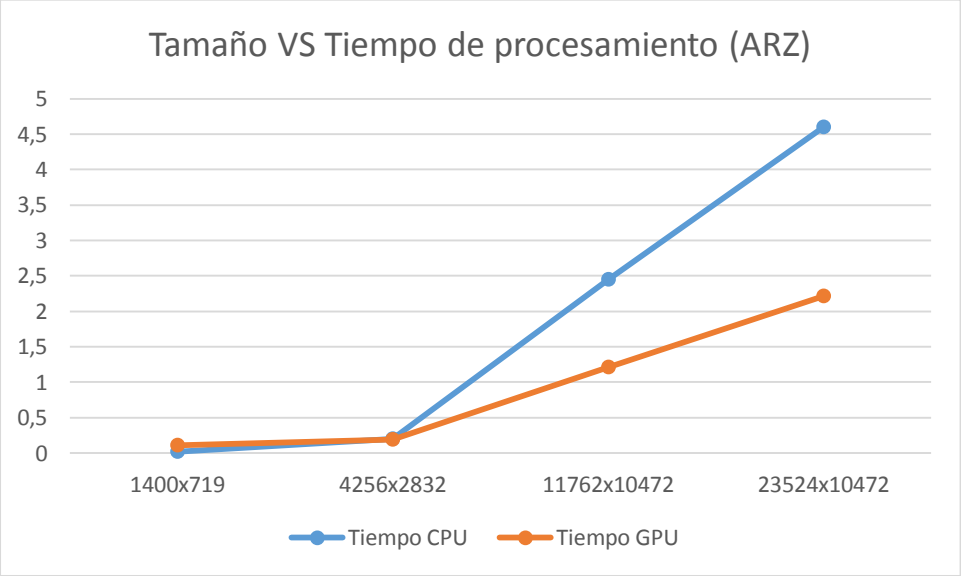

Figura 29. Tamaño vs Tiempo de procesamiento (ARZ). Fuente: Propia

Entre los resultados obtenidos se puede destacar lo siguiente:

- Para resoluciones pequeñas, el algoritmo secuencial es más rápido que el paralelo (speedup menor que uno), ya que para este último en estos casos el mayor tiempo se pierde transfiriendo y recibiendo los datos a/de la GPU.
- El speedup aumenta en consonancia con el tamaño de la imagen, hasta que llega un punto en el que tiende a ser constante.
- En la figura 29 se puede observar como hay un cierto tamaño para el que ambos algoritmos obtienen los mismos tiempos de procesamiento.
- En dicha figura, también se puede intuir como el aumento del tiempo de procesamiento, a partir de tamaños que se pueden considerar grandes siguen una función lineal, si bien, la pendiente para el caso secuencial es bastante más pronunciada que para el caso paralelo.

4.2. ALGORITMO DE CALIBRACIÓN ABSOLUTA (ACA)

RESOLUCIÓN DE LA IMAGEN	TIEMPO DE PROCESAMIENTO (S)		SPEEDUP
	CPU	GPU	CPU/GPU
1400x719	0,125	0,265	**0,472x**
4256x2832	1,591	0,281	5,660x
11762x10472	16,380	1,607	10,193x
23524x10472	32,838	3,151	**10,421x**

Tabla 6. Tiempos de procesamiento para el algoritmo de calibración absoluta

La tabla 6 muestra los resultados obtenidos en ambas versiones para el caso del algoritmo de calibración absoluta.

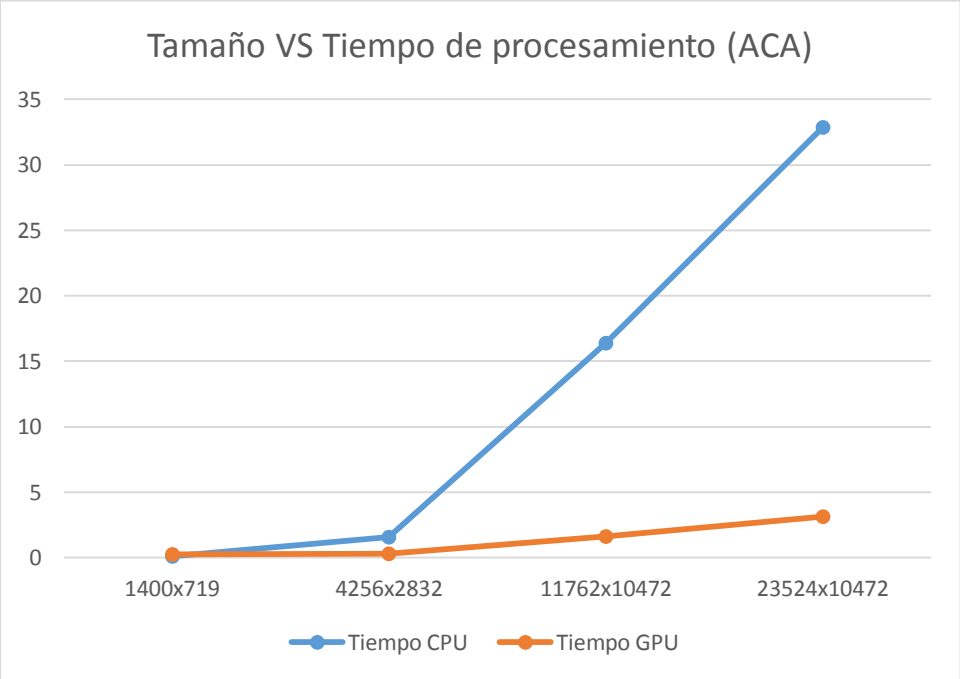

Figura 30. Tamaño VS Tiempo de procesamiento (ACA). Fuente: Propia

Analizando los resultados obtenidos se puede destacar:

- Al igual que en el caso anterior, para resoluciones pequeñas, el algoritmo secuencial es más rápido que el paralelo (speedup menor que uno), ya que para este último el mayor tiempo se pierde transfiriendo y recibiendo los datos a/de la GPU.
- En este caso, el speedup sufre un extraordinario aumento conforme incrementa el tamaño de la imagen, aunque también se intuye un límite en el cual permanece constante.
- En la figura 30, al igual que pasaba para el algoritmo de reconocimiento de zonas, se puede ver como hay un cierto tamaño para el que ambos algoritmos obtienen los mismos tiempos de procesamiento, aunque cabe destacar que, en este caso, dicho tamaño es menor.

- En la figura mencionada, también se puede intuir como el aumento del tiempo de procesamiento en ambos casos sigue una función lineal, al igual que en el algoritmo de reconocimiento de zonas, si bien, la pendiente para el caso secuencial es bastante más pronunciada que para el caso paralelo.

4.3. COMPARACIÓN ENTRE ALGORITMOS

ALGORITMO	RESOLUCIÓN DE LA IMAGEN	TIEMPO DE PROCESAMIENTO (S)		SPEEDUP
		CPU	GPU	CPU/GPU
Reconocimiento de zonas	1400x719	0,016	0,109	**0,147x**
	4256x2832	0,202	0,187	1,080x
	11762x10472	2,449	1,216	2,014x
	23524x10472	4,602	2,215	2,078x
Calibración absoluta	1400x719	0,125	0,265	0,472x
	4256x2832	1,591	0,281	5,660x
	11762x10472	16,380	1,607	10,193x
	23524x10472	32,838	3,151	**10,421x**

Tabla 7. Comparación de tiempos entre algoritmos

En la tabla 7 se muestra, a modo de resumen, los resultados obtenidos para ambos algoritmos.

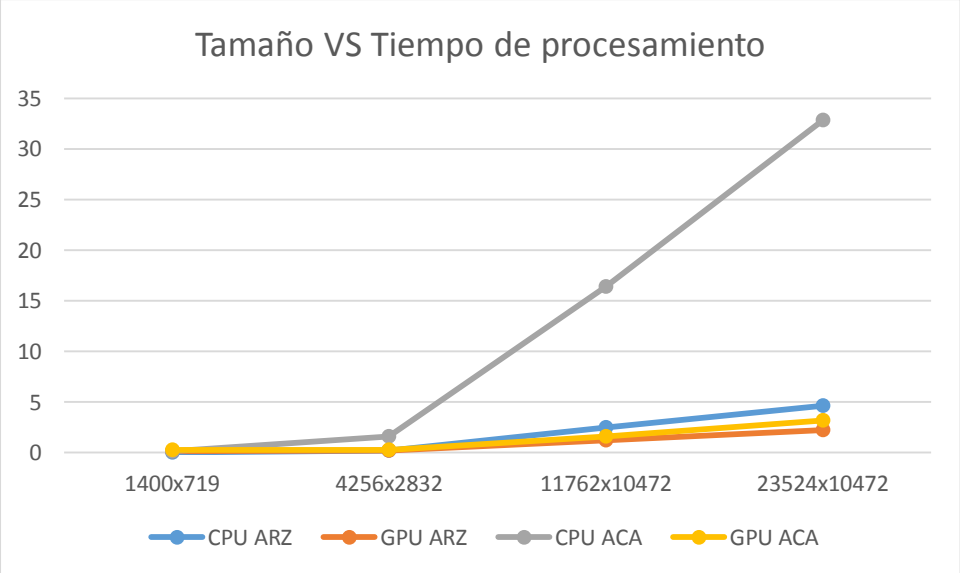

Figura 31. Tamaño VS Tiempo de procesamiento. Fuente: Propia

Basándonos en los tiempos de procesamiento, vemos que en la GPU se obtienen muy parecidos en ambos algoritmos, inclinándose levemente la balanza a favor del algoritmo de reconocimiento de zonas. Esto es debido a que, mientras que en este último se comparan las muestras de las tres bandas con sus respectivos umbrales en la misma sentencia, en el algoritmo de calibración absoluta se realiza el cálculo del valor de cada muestra en una función diferente, además, desde esta función se llama a otra que prueba si se ha saturado el valor obtenido, es decir, si es mayor que el máximo permitido, y en su caso asigna dicho valor máximo a esa muestra.

Por otra parte, como se puede verificar en la figura 31, los tiempos de procesamiento en la CPU del algoritmo de calibración absoluta son significativamente peores que para el de reconocimiento de zonas. Estos tiempos pueden variar dependiendo de las imágenes y de la CPU utilizada.

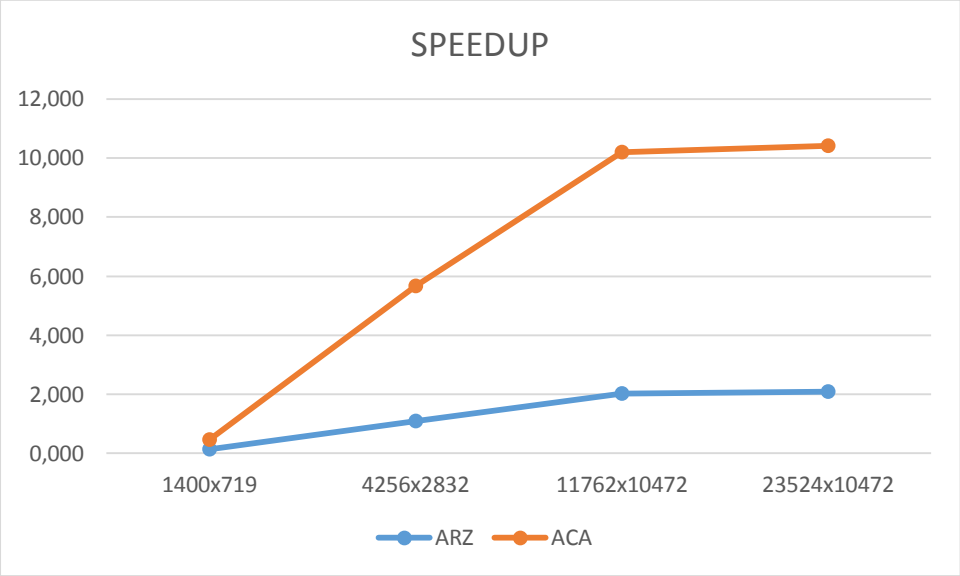

Figura 32. Speedup. Fuente: Propia

En la figura 32 se presenta la comparación del speedup del algoritmo de reconocimiento de zonas (línea naranja) y su homónimo de calibración absoluta (línea azul). En ambos casos sufre un rápido crecimiento en la transición de imágenes de tamaño pequeño a mediano, sin embargo, para imágenes de gran tamaño, el valor permanece constante.

Analizando los resultados obtenidos se puede advertir que, en igualdad de condiciones, CUDA obtiene mejor rendimiento realizando operaciones aritméticas que lógicas.

De cualquier manera, es importante reseñar el ahorro de tiempo que implica el uso de un algoritmo paralelo implementado en CUDA, ya que, para imágenes de gran tamaño, como se puede corroborar en la figura 32, procesa los datos hasta diez veces más rápido, es decir, por cada minuto de procesamiento secuencial, ahorra 54 segundos.

Téngase en cuenta que, como se ha mencionado anteriormente, la GPU utilizada para el procesamiento de las imágenes en este trabajo pertenece a la primera generación de CUDA, por lo que estos resultados podrían mejorarse mediante la utilización de cualquier GPU de una generación posterior.

4.4. ALGORITMO DE FILTRO DE MEDIANA

RESOLUCIÓN DE LA IMAGEN	VENTANA	TIEMPO DE PROCESAMIENTO (S)		SPEEDUP
		CPU	GPU	CPU/GPU
1400x719	3	2,106	0,530	**3,974x**
	5	9,796	4,103	2,3875x
	7	28,735	17,722	1,621x
4256x2832	3	26,021	6,973	3,732x
	5	118,17	215,451	**0,548x**

Tabla 8. Tiempos de procesamiento para el algoritmo filtro de mediana

La tabla 8 muestra los tiempos de procesamiento obtenidos para el filtro de mediana implementado.

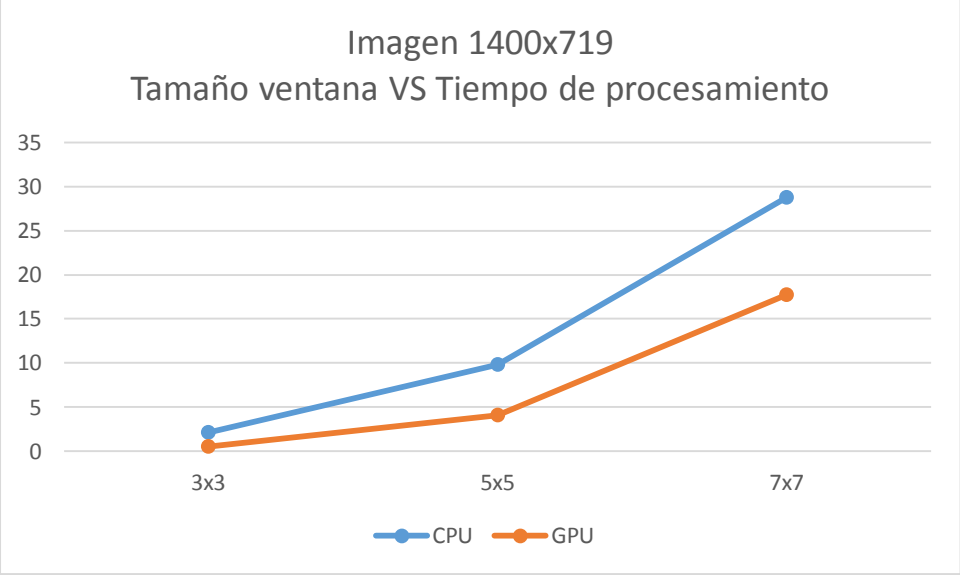

Figura 33. Imagen 1400x719. Tamaño ventana VS Tiempo de procesamiento

Como muestra la figura 33, en la imagen con resolución 1400x719 CUDA obtiene mejores tiempos de procesamiento para cada uno de los distintos tamaños de ventana expuestos, aunque ambas implementaciones tienen una pendiente de crecimiento muy parecida, lo que indica que la GPU no obtiene la ganancia de tiempo que se esperaba con respecto a la GPU.

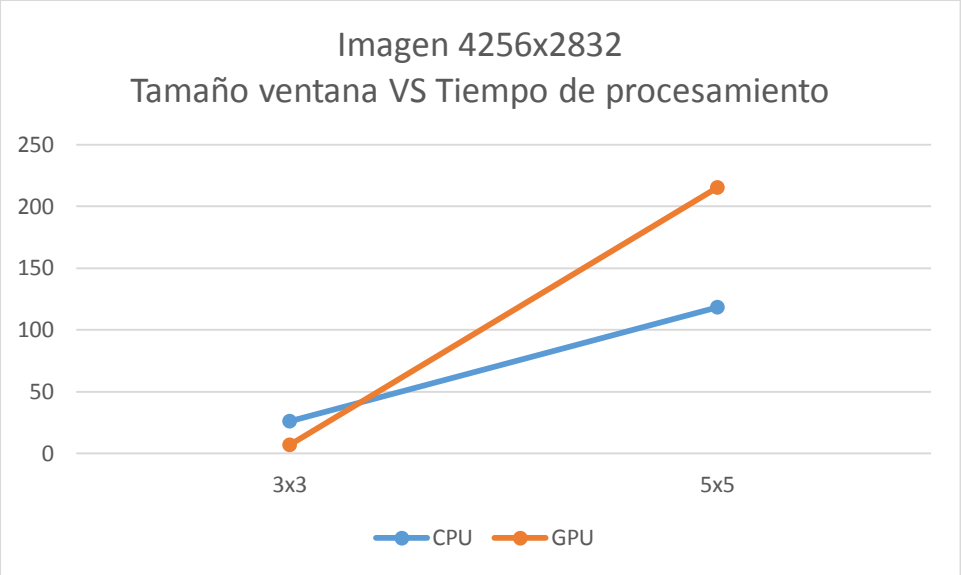

Figura 34. Imagen 4256x2832. Tamaño ventana VS Tiempo de procesamiento

Por otro lado, en la figura 34 se incluyen los resultados obtenidos tras la aplicación del filtro de mediana, con los distintos tamaños de ventana, sobre la imagen con resolución 4256x2832.

En ellos se observa como para el caso de un tamaño de ventana de 3x3, el algoritmo ejecutado en la GPU obtiene mejores resultados que su homónimo en la CPU. Sin embargo, para una ventana de 5x5 es la CPU quien realiza la tarea en menos tiempo.

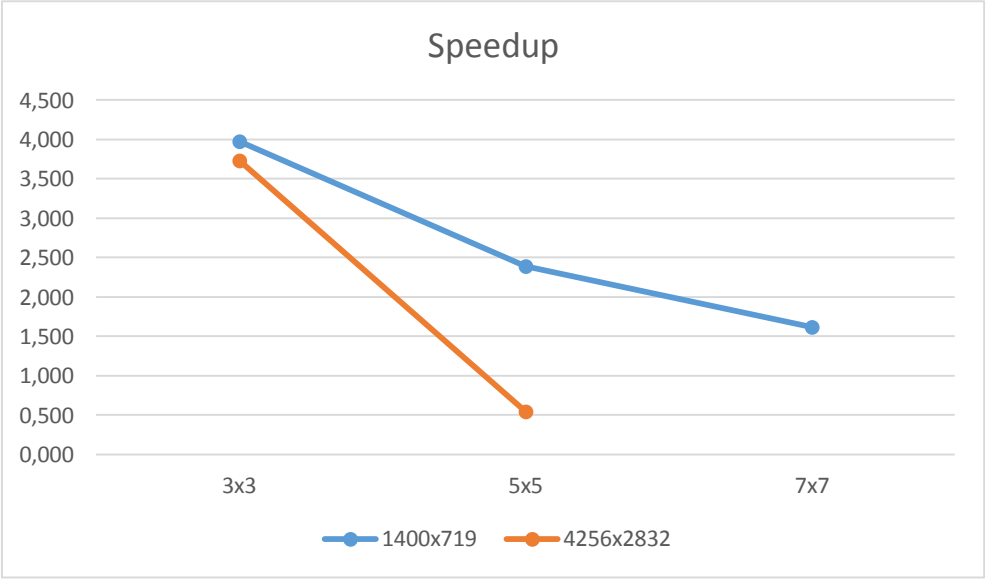

Figura 35. Comparación de speedup entre imágenes

Las explicaciones anteriores se pueden confirmar en la figura 35. En ella se advierte la tendencia de crecimiento negativa del speedup, llegando incluso a ser menor que uno para el caso de una ventana de 5x5 en la imagen de mayor resolución, momento en el que los resultados en la CPU son mejores que en la GPU.

Dicha tendencia negativa también se hace presente en el speedup de la imagen de menor tamaño, lo que indica que si se sigue ampliando el tamaño de la ventana, en algún momento se alcanzarán mejores resultados en la CPU que en la GPU.

Estos resultados dejan entrever las limitaciones de la tarjeta gráfica en la que se han realizado las ejecuciones de los algoritmos. Esto se debe a las exigencias de memoria que requiere este algoritmo, ya que, como se ha indicado con anterioridad, necesita alojar, además de los vectores de entrada y salida, el vector donde se ordenan los píxeles colindantes al que se quiere calcular, lo que implica la reducción del número de threads que pueden usarse durante la ejecución y el aumento del número de transacciones de datos entre ambas unidades de procesamiento (central y gráfico), con la correspondiente bajada de rendimiento en la ejecución del algoritmo.

De lo anterior, se puede deducir que se necesita una tarjeta gráfica con mayor capacidad de cómputo para obtener mejores resultados en la GPU que en la CPU.

5. CONCLUSIONES

El procesamiento paralelo es fundamental a la hora de realizar procesamiento de imágenes de gran tamaño. A la vista de los resultados obtenidos en la realización de este proyecto, CUDA es una gran alternativa a la hora de afrontar un desarrollo paralelo, ya que:

- Frente a un algoritmo secuencial, reduce el tiempo de procesamiento de manera notoria, gracias al aprovechamiento de las capacidades de la GPU.
- Se trata de una extensión del lenguaje C, por lo que su sintaxis es conocida.
- Su rendimiento es mayor a medida que aumenta el tamaño de la imagen, salvo casos excepcionales.
- Obtiene grandes prestaciones tanto para operaciones lógicas como aritméticas, si bien en estas últimas, la reducción de tiempo con respecto a un algoritmo secuencial es ostensiblemente mayor.

A su vez, CUDA presenta una serie de desventajas, entre las que destacan:

- Únicamente puede ser utilizado en tarjetas gráficas NVIDIA.
- Los tiempos de procesamiento para imágenes de pequeño tamaño son peores que en los algoritmos secuenciales, salvo en caso de algoritmos complejos.
- Para algoritmos con alta complejidad, se necesita una GPU con una elevada capacidad de cómputo, de lo contrario, se obtendrán peores resultados que en una CPU convencional.
- Su depuración es enormemente costosa, ya que:
    - En la mayor parte de las ocasiones, muestra errores muy genéricos que no dan suficientes pistas para la resolución de los problemas.
    - Al ejecutarse en la tarjeta gráfica, no permite depuración instrucción a instrucción, hecho que complica aún más este aspecto.

Como conclusión final, destacar que la herramienta implementada es intuitiva y sencilla de utilización.

6. LÍNEAS FUTURAS

Hay varias líneas de futuro sobre las que esta investigación puede seguir su curso, la más obvia es la implementación de nuevos algoritmos tanto de procesamiento como de filtrado de imágenes, que pueden ser de mayor o menor dificultad.

Otra de las líneas a seguir es la aplicación de esta metodología a otros tipos de imágenes de diferente resolución, tanto espacial como espectral para, de esta manera, ampliar las zonas del espectro en las que se puede aplicar los algoritmos desarrollados.

De la misma forma, otra de las alternativas que se presenta, dentro del abanico de posibilidades, es la aplicación de la herramienta a las imágenes generadas por otro tipo de sensores de Teledetección, como pueden ser los sensores activos.

Por último, también se puede trabajar en la implementación de esta herramienta en un formato admitido por los sistemas operativos UNIX, ya que la implementada para la realización de este trabajo solamente puede ejecutarse sobre plataformas Windows.

7. REFERENCIAS

Libros

- Chuvieco, E. 2010. Teledetección ambiental. La observación de la Tierra desde el espacio. Editorial Ariel Ciencias.
- Mather, P y Koch, M. 2011. Computer Processing of Remotely-Sensed Images: An Introduction. 4ª edición. Editorial John Willey & Sons.
- Sobrino, J. A. 2000. Teledetección. Editor Sobrino, J. A. Universidad de Valencia.

Artículos

- Park, I. K., Singhal, N., Lee, M.H., Cho, S., Kim, C. W. 2011. Design and performance evaluation of image processing algorithms on GPUs. *IEEE transactions on parallel and distributed systems*, vol. 22(1), 91-104.

Páginas web consultadas:

- http://docs.nvidia.com/cuda/#axzz3UMNxaWFm
- http://www.libtiff.org/
- https://www.visualstudio.com/
- https://msdn.microsoft.com
- http://sistema-de-reservas-es.bitpalast.net/pr/i
- http://www.zeiss.es/corporate/es_es/home.html
- http://www.uncoma.edu.ar/